

The background of the slide features a complex, abstract pattern of overlapping, curved, light-colored panels, possibly made of paper or fabric, creating a sense of depth and texture. The lighting is dramatic, with strong highlights and shadows emphasizing the folds and curves of the material.

Alex Lawrence

Implementing DDD, CQRS and Event Sourcing

Implementing DDD, CQRS and Event Sourcing

Alex Lawrence

This book is for sale at <http://leanpub.com/implementing-ddd-cqrs-and-event-sourcing>

This version was published on 2021-04-15



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2017 - 2021 Alex Lawrence

Tweet This Book!

Please help Alex Lawrence by spreading the word about this book on [Twitter](#)!

The suggested hashtag for this book is [#ddd-cqrs-es](#).

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

[#ddd-cqrs-es](#)

This book is dedicated to my wonderful wife and kids. I love you.

Contents

Preface	i
Version & Feedback	i
Style of this book	i
Executing the code	iv
About the author	v
Formatting and highlighting	v
Chapter summary	vi
Chapter 1: Domains	1
Technological and business domains	1
Domain Experts	3
Subdomains	4
Identification of Subdomains	6
Sample Application: Domains	8
Chapter 2: Domain Models	12
Structure and components	12
Ubiquitous Language	14
Domain Modeling	17
Model representations	18
Sample Application: Domain Models	19
Chapter 3: Bounded Contexts	26
Relation to Domains	26
Model and context sizes	27
Contexts and language	30
Technological boundaries	31
Context Maps	33
Sample Application: Bounded Contexts	34

CONTENTS

Chapter 4: Software Architecture	38
Common software parts	38
Layered Architecture	41
Onion Architecture	42
Approach for this book	42
Sample Application: Software architecture	43
Chapter 5: Code quality	46
Model binding	46
Readability	48
Behavior and state	52
Dealing with dependencies	55
Command-Query-Separation	58
Chapter 6: Value Objects, Entities and Services	64
Value Objects	64
Entities	71
Domain Services	82
Invariants	83
Sample Application: Domain Model implementation	85
Chapter 7: Domain Events	99
Relation to Event-driven Architecture	99
Naming conventions	100
Structure and content	100
Distribution and processing	104
Sample Application: Event-based integration	118
Chapter 8: Aggregates	122
Transactions	122
Structure and access	123
Concurrency	127
Design considerations	136
Eventual Consistency	143
Sample Application: Aggregates	149
Chapter 9: Repositories	161
Domain Model emphasis	161
Design and implementation	162

CONTENTS

Optimistic Concurrency	170
Interaction with Domain Events	181
Sample Application: Repositories and event publishing	194
Chapter 10: Application Services	208
Service design	208
Use case scenarios	213
Transactions and Processes	214
Cross-cutting concerns	225
Authentication and Authorization	234
Sample Application: Application Services	243
Chapter 11: Command Query Responsibility Segregation	258
Architectural overview	258
Write and Read Model	260
Read Model synchronization	269
Commands and Queries	279
Command and Query Handlers	284
Sample Application: CQRS	293
Chapter 12: Event Sourcing	315
Architectural overview	315
Event-sourced Write Model	319
Event Store	327
Read Model Projection	338
Domain Event publishing	347
Sample Application: Event Sourcing	349
Chapter 13: Separate executable programs	367
Program layouts	368
Context-level communication	369
Remote use case execution	370
Remote event distribution	375
Sample Application: Separate executable programs	378
Chapter 14: User Interface	387
HTTP file server	387
Task-based UI	390
Optimistic UI	397

CONTENTS

Reactive Read Models	401
Components and composition	419
Sample Application: User Interface	427
Conclusion	447
Apply the useful parts	447
The use of frameworks	447
Appendix A: Static types	449
Value Objects, Entities and Services	449
Events	455
Event-sourced Write Model	458
Dependency Inversion	461
Commands and Queries	464
Sample Application: TypeScript implementation	465
Appendix B: Going into production	467
Identifier generation	467
Containerization and Orchestration	468
Event Store	470
Read Model store	473
Message Bus	475
HTTP server	477
Sample Application: Going into production	478
Bibliography	484

Preface

This book explains and illustrates how to implement Domain-Driven Design, Command Query Responsibility Segregation and Event Sourcing. The goal is to build software that is behavior-rich, event-based, problem-centric, reactive, scalable and well-designed. **Domain-Driven Design** is a concept that focuses on the problem space and its associated knowledge areas. **Command Query Responsibility Segregation** separates a software into a write side and a read side. **Event Sourcing** is an architectural pattern that represents state as a sequence of immutable events. The concepts are explained in theory and put into practice with standalone examples and a Sample Application. This is done without third-party technologies. The book comes with a source code bundle and supports interactive execution. All code is written in **JavaScript** and uses **Node.js** as backend runtime.

Version & Feedback

Book version: 1.3.3

If you have any questions, suggestions or problems to report, please write an e-mail or visit the book's Leanpub Forum. The contact e-mail address is mail@alex-lawrence.com and the Leanpub forum can be found [here](#). All feedback is highly appreciated.

Style of this book

The primary focus of this book is the application and the implementation of concepts. Therefore, the purely theoretical parts are generally concise. The covered topics are illustrated extensively with a large amount of examples and code. Selected conceptual parts are also discussed in greater detail. Apart from **Node.js** and **JavaScript**, the book's main content does not utilize or explain specific frameworks or technologies. For functionalities that require persistence or inter-process communication, exemplary implementations are provided that directly work with the filesystem. This includes Repositories, the Event Store, Read Model stores and a remote event distribution. The goal is to convey a deeper understanding of the according concepts. For production purposes, these implementations can be replaced with suitable technologies. This procedure is exemplified in Appendix B.

Why Node.js and JavaScript?

There are multiple reasons for using Node.js as runtime platform and JavaScript as programming language. One is my personal long-term experience with them. Also, two projects in which I applied CQRS and Event Sourcing made use of both technologies. Another reason for JavaScript is that it is a very widespread language. Even more, its syntax can look similar to other popular languages, such as Java, especially when using classes. Furthermore, this book uses JavaScript for the frontend and shares some code with the backend. One specific reason for Node.js is its simplicity for certain use cases, such as when operating an HTTP server. Finally, JavaScript helps to keep the code examples concise through specific language features such as arrow functions or destructuring assignments.

Programming paradigms

The content and the code examples in this book compromise a combination of imperative, declarative, object-oriented and functional programming. However, the majority of the implementations apply the object-oriented paradigm. Also, classes are used extensively as well as private fields and private methods for strong encapsulation. Still, over the course of the book, the domain-related implementations transition towards a more functional style. Also, certain infrastructural functionalities apply selected principles of it wherever useful. As the changes come naturally together with introducing specific concepts, there is no need for upfront knowledge in Functional Programming.

Type checking and validation

The source code provided with this book only makes selective use of type checking and data validation. Since JavaScript is a dynamically typed language, type checks can only happen at runtime. Also, the available technical possibilities are rather limited. In contrast, data validation itself is not fundamentally different from most other languages. However, due to the lack of static types, it can be more cumbersome. This book uses both mechanisms only for specific examples and for Sample Application parts where data integrity is crucial. This means, verification checks are primarily implemented for domain-related components, for Domain Events as well as for Commands and Queries. Note that this approach does not imply any opinion about the general importance of type checking and data validation.

Required technical knowledge

Working through the book requires advanced knowledge in JavaScript and in Node.js. Many of the provided implementations use language features introduced with more re-

cent ECMAScript standards. The following lists show which newer language constructs, which JavaScript APIs and which Node.js APIs are assumed to be known. Every entry is accompanied by a link, pointing to either a documentation page or a tutorial. Generally, it is not necessary to read through all of them. Many functionalities can be understood from the context they are used in. The only exceptions are **Promises** and **async/await**, which provide an alternative to callbacks for asynchronous programming. As those concepts heavily influence the way of writing code, getting familiar with them should be done upfront.

Newer JavaScript language constructs:

- Arrow functions
- Async functions
- Await operator
- Classes, specifically private class fields
- const, let
- Destructuring assignment
- JavaScript Modules
- Shorthand property names
- Spread Syntax
- Template literals
- Throwing and handling Errors

Used JavaScript APIs:

- Array functions such as `map()`, `reduce()`, `forEach()`, `includes()`, `filter()` and `flat()`
- Console
- JSON
- Map
- Object functions such as `defineProperty()`, `defineProperties()`, `assign()` and `freeze()`
- Promises
- Proxy
- URLSearchParams

Used Node.js APIs:

- `crypto.createHash()`
- `fs.watch`

- many of the standard operations of `fs.promises`
- `http` functions such as `createServer()`, `get()`, `post()`
- various `path` functions
- `Readable Streams` and `Transform Streams`
- `url.parse()`
- `child_process.spawn()`

Executing the code

This book provides an easy possibility to execute most of the shown code examples when reading on a computer. The included bundle contains a Code Playground than can execute Node.js code and display its output. Almost every implementation in the book is accompanied by a hyperlink labeled as “run code” or “run code usage”. Clicking such a link opens a web page with an editor and an output window. Please note that **this utility executes Node.js on your local machine**. Therefore, you are responsible for verifying the safety of each program. The only requirement is to have a recent version of Node.js installed. Running the playground is done by executing `npm start` inside the bundle root directory. The software starts an HTTP server on port 8080.



.data directory

Starting from Chapter 9, many code examples and Sample Application implementations save data to the filesystem. Every created file and subdirectory is put within a directory called “`.data`”, which itself is placed at the bundle root. This data container can be deleted at any time without influencing the functionality of associated code.

The complete source code that is bundled with the book is licensed under the **MIT License**. This means, you are free to re-use every contained functionality for your own projects. However, please refrain from publishing the complete bundle, as it represents an essential part of the work for this book. Also, note that the code is primarily for illustration purposes. While the implementations fulfill the respective functional requirements, they are not necessarily ready to be used in production environments. Amongst other things, the code may miss essential validations and can be vulnerable to security-related attacks.

About the author

I am a software developer with knowledge and experience in architecture, automation, backend, frontend, operations, teaching, technical leadership and testing. Since 2007, my professional focus lies on full-stack web development. In most projects, I use JavaScript as language and Node.js as backend runtime. Wherever useful, I apply selected parts of DDD. The architectural patterns I am most interested in are Event-driven Architecture, CQRS and Event Sourcing. For the frontend, I personally favor to use native technologies, such as Web Components. Professionally, I also work with various libraries and tools. Most recently, I started learning Rust as new programming language and picked up selected concepts of Functional Programming. Since many years, I am a strong supporter for Free/Libre Open Source Software.

Ever since the age of 10, I have been interested in computers. Experimenting with BASIC on a C64, programming with Turbo Pascal and failing to self-teach C++ accompanied my early adolescence. Only much later when studying Informatics, I deep-dived into the theory and learned many other programming languages. My first job introduced me to advanced concepts and patterns, including CQRS and Event Sourcing. At some point, I quit my job and became a freelancer. Simultaneously, I launched a mouse tracking product that naturally applied Event Sourcing. Also, I joined a startup developing a collaborative software with strong focus on DDD, CQRS and Event Sourcing in Node.js. In this project, I had the chance to deepen my theoretical knowledge and apply many concepts practically.

Formatting and highlighting

The book uses certain formatting and highlighting to either categorize or emphasize information. The names of acknowledged concepts are always capitalized, such as Entity or Domain Model. Words written in **bold** either indicate that a term is defined and/or explained, or simply emphasize an important text passage. References to code constructs such as classes, functions, keywords or variables are displayed in monospaced font. Complementary content is placed into separately positioned paragraphs, visually highlighted with an icon. Typically, this is an “i” enclosed in a circle. Wherever it makes sense, lists and tables are used to display information. Apart from plain text, this book includes simple drawings and numerous code examples. The displayed code is not always complete. Rather, it focuses on illustrating specific facts.



Complementary content

This is an example paragraph of complementary information. It is not required to be read in order to understand the main content.

Chapter summary

The chapter order in this book is determined by what makes most sense with regard to building the Sample Application. As a consequence, the topics are laid out in a way they can build upon each other. Amongst other things, this also reduces the likelihood of referring to terms before they are explained. Every chapter follows the same structure. The main part explains the respective concepts and illustrates them either with drawings or code examples. At the end of each chapter, the discussed concepts are applied to the Sample Application. This is done by describing the working steps to take and by showing the according drawings and code. An exception to this format is Chapter 5, which does not contain a Sample Application section.

Chapter 1

Chapter 1 introduces the concept **Domains** and defines it in the context of DDD as problem-specific knowledge fields. The explanation incorporates the importance of always focusing on the original problems to solve. Different categories of knowledge areas and their significance are compared to each other. This is followed by the definition of **Domain Experts**, which represent the primary source of relevant information in each project. For an adequate subdivision of knowledge areas, the concept **Subdomains** is introduced. This is accompanied by a description of the different types **Core Domain**, **Supporting Subdomain** and **Generic Subdomain**. A brief illustration on how to identify Subdomains provides practical guidance. At the end of the chapter, the Domains and the Subdomains for the Sample Application are identified.

Chapter 2

Chapter 2 covers the concept **Domain Models**, which are sets of knowledge abstractions focused on solving specific problems. The chapter starts with describing their typical structure and components, and explains why such abstractions should incorporate verbs, adjectives and adverbs. Also, the relation to Domain Expert knowledge is clarified. As next

step, the concept **Ubiquitous Language** is introduced, which promotes a linguistic system to unify communication and eliminate translation. Afterwards, different representation possibilities for Domain Models are described and compared to each other, such as drawings and code experiments. This is followed by a section on **Domain Modeling**, which is the process of creating structured knowledge abstractions. As last step, the Sample Application Domain Model is created and expressed in multiple different ways.

Chapter 3

Chapter 3 focuses on the concept **Bounded Contexts**, which represent conceptional boundaries for the applicability of Domain Models. First, the concept is differentiated from Domains and Domain Models. Then, multiple context sizes and their implications are compared to each other. This includes large unified interpretations and smaller boundaries that align with Subdomains. Afterwards, the relation between a Bounded Context and a Ubiquitous Language is clarified. Also, it is explained why contexts are first and foremost of conceptual nature, but still commonly align with technological structures. The relationship and integration patterns **Open Host Service**, **Anti-Corruption Layer** and **Customer-Supplier** are discussed briefly. This is followed by the concept **Context Map** for visualizing conceptional boundaries. Finally, the chapter illustrates the definition of Bounded Contexts for the Sample Application.

Chapter 4

Chapter 4 deals with **Software Architecture**, which is a high-level structural definition of a software and its parts. Since this topic is very broad, the focus is narrowed down to common architectural patterns that fit well with DDD. The chapter starts with describing the typical parts of a software, consisting of **Domain**, **Infrastructure**, **Application** and **User Interface**. This is followed by describing and comparing the patterns **Layered Architecture** and **Onion Architecture**, which share many fundamental principles. Afterwards, it is explained and justified what approach the book and its implementations follow. Also, the section includes an explanation on how to invert dependencies between software layers by using abstractions. The chapter ends with defining the Software Architecture and the resulting directory layout for the Sample Application.

Chapter 5

Chapter 5 describes and illustrates selected concepts for a high code quality with focus on Domain Model implementations. First, it explains the importance of establishing a

binding between the Domain Model and its implementation, together with other artifacts. Afterwards, refactoring is described as functional refinement in the context of DDD and is differentiated from pure technical improving. Next, code readability is discussed and its characteristics are broken down into quantifiable aspects. The importance of combining related state and behavior is emphasized and exemplified by creating meaningful units with high cohesion. This is followed by an explanation on how to deal with dependencies and how to apply **Dependency Inversion**. The final concept of the chapter is **Command-Query-Separation**, which promotes a separation of state modifications and computations.

Chapter 6

Chapter 6 provides a number of tactical patterns as essential building blocks for a Domain Model implementation. First, **Value Objects** are introduced as a way to design a descriptive part of a Domain without a conceptual identity. This is accompanied by an illustration of the key characteristics Conceptual Whole, Equality by Values and Immutability. Afterwards, **Entities** are described as possibility to design unique, distinguishable and changeable Domain Model components. This includes an explanation of identities and how to generate reliable identifiers using **UUID**. Next, **Domain Services** are discussed for expressing stateless computations and encapsulating external dependencies through abstractions. Afterwards, **Invariants** are presented as a way to transactionally ensure consistent state. Finally, all patterns are applied to create a useful first Sample Application Domain Model implementation.

Chapter 7

Chapter 7 presents the concept of **Domain Events**, which represent structured knowledge of specific meaningful occurrences in a Domain. The chapter starts with explaining their relation to Event-Driven Architecture. This is followed by describing common naming conventions for creating expressive event types based on a Ubiquitous Language. Afterwards, the standard structure and content of events are discussed, including a differentiation between specialized values and generic information. Next, it is described how the distribution and processing of Domain Events work. This includes the implementation of an in-memory **Message Bus** and an **Event Bus**. Also, important challenges of a distribution process are discussed briefly. At the end of the chapter, Domain Event notifications are used to integrate different context implementations of the Sample Application.

Chapter 8

Chapter 8 introduces **Aggregates** to establish transactional consistency boundaries, which are essential for concurrency and persistence. The first section contains an explanation of **transactions** and a breakdown of their characteristics Atomicity, Consistency, Isolation and Durability. This is followed by discussing the structural possibilities of Aggregates, the **Aggregate Root** and the management of individual component access. Afterwards, the principle of **Concurrency** is explained and illustrated in detail, together with Concurrency Control. Then, the most important Aggregate design considerations are described, which focus on component associations, invariants and optimal size. **Eventual Consistency** is introduced as mechanism to synchronize distributed information across consistency boundaries in a non-transactional fashion. Finally, the Sample Application is analyzed and refactored in order to achieve a useful Aggregate design.

Chapter 9

Chapter 9 describes the concept of **Repositories** for enabling persistence in a meaningful way to a Domain Model. First, the chapter underlines the emphasis on the Domain Model as opposed to technological aspects. Then, a basic Repository functionality is illustrated that consists of saving, loading and custom querying. Also, the influences of persistence support on a Domain Model expression are exemplified. The next part explains **Optimistic Concurrency** and promotes version numbers as implementation. Furthermore, it describes automatic retries and conflict resolution for concurrency conflicts. Afterwards, the chapter focuses on the interaction with Domain Events and provides different publishing approaches. One promotes extending Repositories and the other one recommends separating concerns. Lastly, the Sample Application is refactored for persistence support together with reliable Domain Event publishing.

Chapter 10

Chapter 10 illustrates **Application Services**, which are the responsible part for executing use cases, managing transactions and handling security. The first section describes and compares two approaches for their overall design and implementation. This is followed by an explanation of different possible service scenarios, ranging from simple Entity modifications to specialized stateless computations. Afterwards, the topics transactions and consistency are explained with regard to Application Services and complemented with a detailed example. The subsequent section discusses cross-cutting concerns and utilizes the design patterns **Decorator** and **Middleware** for generic solution approaches. Next, the security-related

concepts **Authentication** and **Authorization** are explained briefly and illustrated with individual examples. The chapter ends with implementing the Application Services for the Sample Application together with exemplary authentication and authorization.

Chapter 11

Chapter 11 explains **Command Query Responsibility Segregation**, which separates a software into a write side and a read side. The chapter starts with an architectural overview to explain the high-level picture as well as the interactions between individual parts. The following section introduces and illustrates the two Domain Layer concepts **Write Model** and **Read Model**. This is followed by a comparison of different approaches for the synchronization of Read Model data. As next step, the message types **Commands** and **Queries** are explained together with recommendations on their naming and structure. Afterwards, **Command Handlers** and **Query Handlers** are presented as a specialized form of Application Services. Finally, all covered concepts are combined and applied in order to create a Sample Application implementation with CQRS.

Chapter 12

Chapter 12 covers the pattern **Event Sourcing**, where state is represented as a sequence of immutable change events. As first part, the chapter provides an architectural overview and describes the overall flow. This is complemented with a clarification on the relation to Domain Events and recommendations on event anatomy. Afterwards, event-sourced Write Models are explained and two different implementation approaches are compared. Then, the concept **Event Store** is introduced for the persistence of event-sourced state. This is followed by a section on **Read Model projections**, which are responsible for computing derived read data. Next, the concept of Event Sourcing is differentiated from Event-Driven Architecture and their combination is explained. The chapter ends with transforming the existing Sample Application code into an implementation with Event Sourcing.

Chapter 13

Chapter 13 describes how to split a software into separate executable programs that operate as autonomous and self-contained runtime units. The chapter starts with an explanation of program layout possibilities. This includes a division of architectural layers as well as the alignment with context implementations. Next, the two main types of context-level communication are described and compared to each other. This is followed by a section on **remote**

use case execution, which includes an HTTP interface factory as example implementation. Afterwards, the idea of **remote event distribution** is discussed and complemented with a filesystem-based implementation. As last step, the chapter separates the Sample Application implementation into multiple programs. This incorporates the introduction of a proxy server that provides a unified HTTP interface to multiple endpoints.

Chapter 14

Chapter 14 focuses on the **User Interface** part in the context of web-based software. As first step, the chapter explains how to serve files to browsers and introduces an HTTP file server. Then, the concept of a **Task-based UI** is introduced and compared to a CRUD-based approach. This is followed by explaining and illustrating the **Optimistic UI** pattern, which can improve user experience through optimistic success indication. Afterwards, the concept of **Reactive Read Models** is described, which helps to build reactive User Interfaces. This includes the introduction of the **Server-Sent Events** web standard. Also, the notion of components and their composition is explained together with the **Custom Elements** technology. The chapter ends with illustrating the implementation of the User Interface for the Sample Application.

Appendix A

Appendix A explains the use of static types and their potential benefits with TypeScript as example. The first part focuses on Value Objects, Entities and Services. This is followed by describing how static types affect the definition and the usage of events. Next, the implications on event-sourced Write Models are discussed. Afterwards, the concept Dependency Injection is revisited. Then, the effects of static types on Commands and Queries are explained. The appendix ends with summarizing the most important aspects of a TypeScript implementation for the Sample Application.

Appendix B

Appendix B deals with using existing technologies for production scenarios. The first part discusses identifier generation. This is followed by a section on **Containerization** and **Orchestration** using Docker and Docker Compose. Next, the Event Store is discussed with the example of EventStoreDB. Then, Read Model stores are covered with Redis as exemplary technology. Afterwards, the Message Bus functionality is exemplified with RabbitMQ. Then, a static file server and a proxy functionality are illustrated with NGINX. Finally, the introduced technologies are used for the Sample Application implementation.

Chapter 1: Domains

Generally speaking, a **Domain** is an area of expertise. Put into more abstract terms, it is a conceptual compound of cohesive knowledge. In the context of Domain-Driven Design (DDD), the term stands for the topical area in which a software operates in. [Vernon, p. 43] describes it as “what an organization does and the world it does it in”. As alternative, [Evans, p. 2] defines it as “the subject area to which the user applies a program”. Another way to think of it, is as the knowledge space around the problems a software is designed to solve. The exact definition of the term can vary slightly, depending on its context. Regardless of such detailed differences, it is always important to identify the distinct knowledge areas a software operates in.

Technological and business domains

Independent of individual experience and progression, the profession of a software developer demands to acquire and apply technological knowledge. This may include topics such as Boolean Algebra, Control Structures, Object-oriented Programming, Processes and Threading, Networking and many more. While these concepts are typically illustrated with concrete examples, they themselves are purely technical and applicable to different situations. Therefore, they can be gathered in a common Domain called “Informatics” (or “Computer Science”). When attempting to solve a non-technological problem with software, the crucial knowledge does not originate from this particular Domain. Rather, it is to be found in the knowledge area that is specific to the original problem itself.

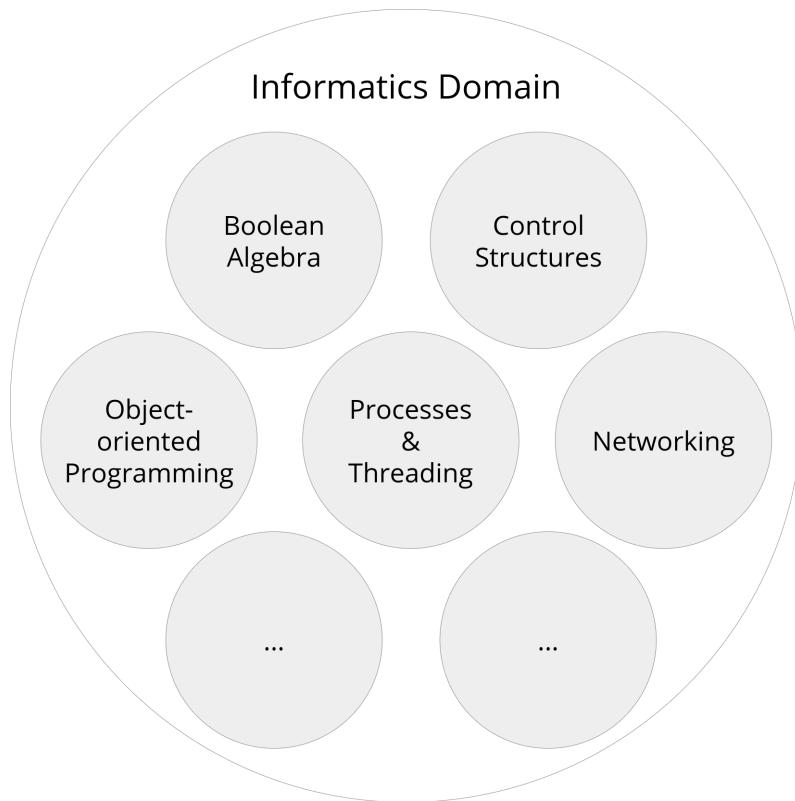


Figure 1.1: Informatics Domain

Business Domain problems cannot be solved adequately with solutions that exclusively belong to technological Domains. One exception is when the business itself is of technological nature, such as when developing a code analysis tool. Without applying the specialized knowledge around a problem, it is virtually impossible to build an adequate software solution. Therefore, it is necessary for involved participants and entities to understand the business Domains they are operating in. This applies to every project, regardless of its size and complexity. There are times when developers get involved in purely technical areas, such as when implementing communication protocols or configuring infrastructure. These fields of knowledge must not be confused with the actual Domain of a company or a product.

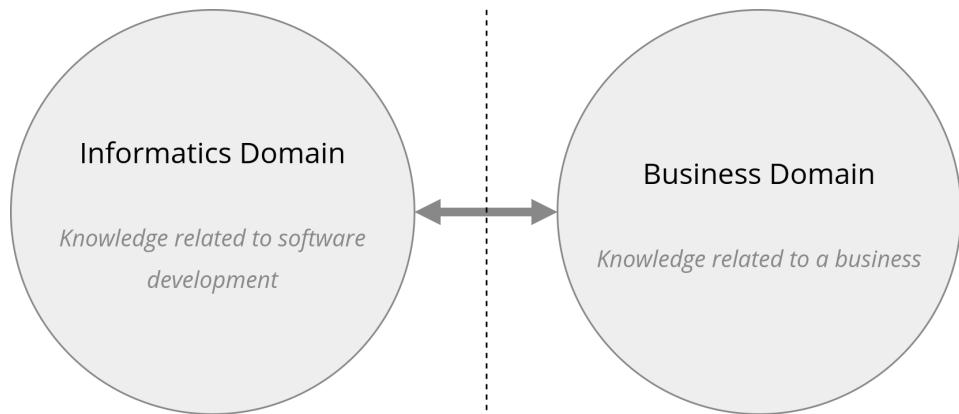


Figure 1.2: Technological vs business Domains

Example: User Experience testing

As example, consider building a software for qualitative user experience testing of websites. The goal is for customers to identify areas and functionalities on their website that cause user frustration. For this purpose, a JavaScript snippet is provided that records user interaction data and sends it to a server. The received information is matched against patterns that indicate user frustration, and the results are presented to the customer. For example, repeated clicks at random positions may be categorized as so-called “rage clicks”. Building such a software requires knowledge in many purely technological areas, such as DOM Events and HTTP. However, the User Experience part is the relevant business Domain. Without understanding and applying its specialized knowledge, it is almost impossible for the software to succeed.

Domain Experts

Domain Experts are the primary source of specialized knowledge that enables to create an adequate software-based solution to a problem. For every project, there should be at least one person with this role. If this is not the case, the relevant Domain knowledge can be acquired by individuals, who then become the experts. In general, Domain Experts should be able to provide helpful answers to most of the arising questions. However, they cannot be expected to know “everything” about a respective Domain. Also, the knowledge may be shared across multiple persons, each with their own specialty. Furthermore, some facts may simply be unknown. In such situations, it makes sense to put effort into discovering the unknown parts and gaining new insights.

Relation to other roles

Typically, a Domain Expert is a secondary role on top of the existing ones inside a software project. Theoretically, the role can be taken on by anyone, be it developers, designers, product managers or agile coaches. Still, it fits best with project members that work closely with the actual customer. Most commonly, this is either a product manager or a product owner. Depending on the project, even the customer themselves can be a Domain Expert. However, such a constellation requires to always strictly distinguish between relevant knowledge and unprocessed feature inquiries. While a customer often understands the Domain in question well, the actual functionalities of a software are best defined by the project team.



Domain Experts in small projects

For small projects, it may be that you as a developer consider yourself the Domain Expert. While this is possible, the setup demands some precaution. Being the developer, the Domain Expert and maybe even the client simultaneously can be problematic. Requirements can become fuzzy or even get abandoned. Make sure to always focus on the actual problem and its solution.

Subdomains

Subdomains are distinguishable knowledge areas that are part of a larger compound. In theory, almost every Domain can be understood as a collection of individual self-contained parts. Likewise, multiple Domains can be grouped together with others into an overarching one. This is mainly a matter of perspective and scope. Amongst other things, it also illustrates the ambiguity of the term “Domain”. On the one hand, it can stand for the whole of something. On the other hand, it can describe a subordinate part. For software projects, the overall Domain may not be a universally meaningful knowledge area. Rather, it can be a collection of parts that may even be unrelated. There are three types of Subdomains, of which each is described as a following subsection.

Core Domain

The **Core Domain** is the knowledge area that is most relevant to the problems a software aims to solve. It should be identified as early as possible. Most of the expertise and effort must be put into this area. With regard to a business, the Core Domain is the key to success.

Ideally, the functionality related to its knowledge area provides competitive advantage over comparable projects. If a software does not excel at its core, the associated business is likely to fail. There are scenarios where the overall Domain of a software exclusively consists of a Core Domain without other Subdomains. This is often the case for projects with a specialized functionality that is meant to be integrated into other software.

Supporting Subdomain

Supporting Subdomains are concerned with knowledge of which some parts are specific to the custom software Domain. Another way to describe them is as a combination of generic knowledge and problem-specific aspects. While these areas are not as important as the Core Domain, they still play a decisive role. There can be an arbitrary number of Supporting Subdomains, each concerned with individual knowledge. This also means that there can be no such part inside a particular Domain. Due to the hybrid character of these topical areas, their existence should always be judged critically. In the worst case, Core Domain components are mistakenly combined with generic knowledge into an artificial Supporting Subdomain.

Generic Subdomain

Generic Subdomains deal with universal knowledge that is not specific to the main problem space of a software. As with Supporting Subdomains, their existence is not mandatory and their count is not restricted. Generally, these parts are good candidates for outsourcing the associated work. The desired functionalities can be developed by external entities. Another option is to use existing third-party software solutions. Note that even when Subdomains are of generic nature, they are still an integral part of the overall Domain. Without appropriately accounting for these secondary parts, a project also risks to fail. The most prevalent Generic Subdomain in software projects is the knowledge area around users, identities, authentication and authorization.

Example: Weather forecasting

Consider building a weather forecasting software. The desired functionality is to provide weather forecasts as well as rain activity reports and weather alerts for specific locations. For the definition of the Domain, all Subdomains must be identified and classified. The Core Domain is the “Weather Forecasting”, which is the key knowledge area of the software. In addition to that, there are two Supporting Subdomains. One is the “Precipitation

Reporting”, which contains the knowledge for building a rain radar. Secondly, the “Weather Alerting” Subdomain is the part that is required for providing weather alerts. Both areas are concerned with aspects that relate to the core. Finally, a Generic Subdomain is the “Location Mapping”, which incorporates the knowledge for converting human-readable locations into geolocations.

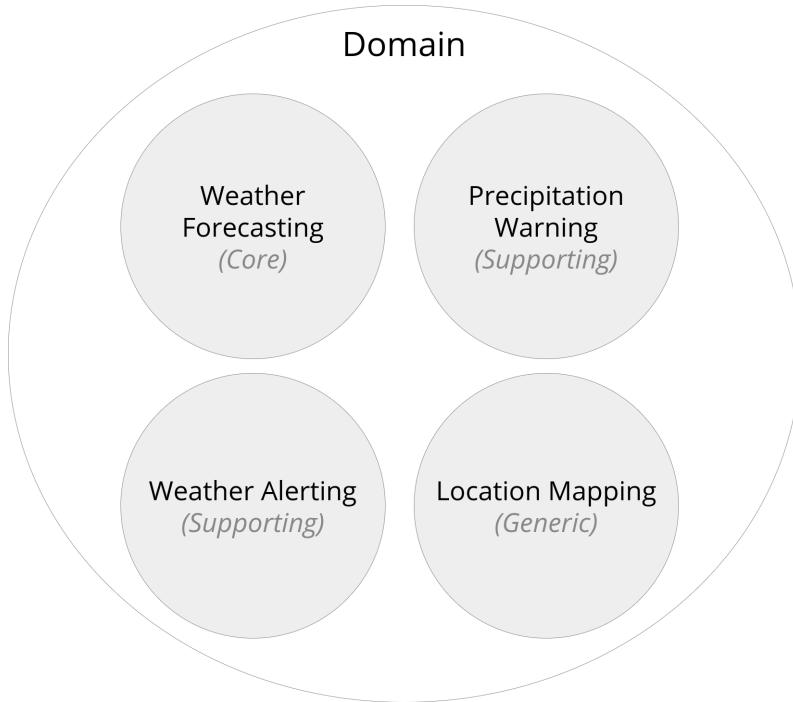


Figure 1.3: Weather Forecasting Domains

Identification of Subdomains

The identification of Subdomains for a software is a fairly subjective process. There are no exact rules on how to perform this step. Consequently, there is no absolute right or wrong in the way of doing it. In fact, it is heavily dependent on the project and the people working on it. Whatever helps to capture the individual knowledge areas at play, should be considered useful. One visualization approach is to draw circles for Subdomains, add in their names and types, and create lines for relationships. This type of artifact does neither have to be created by a single person, nor exclusively by developers. Since the Subdomain definitions are relevant for every project member, the activity is ideally a team effort.

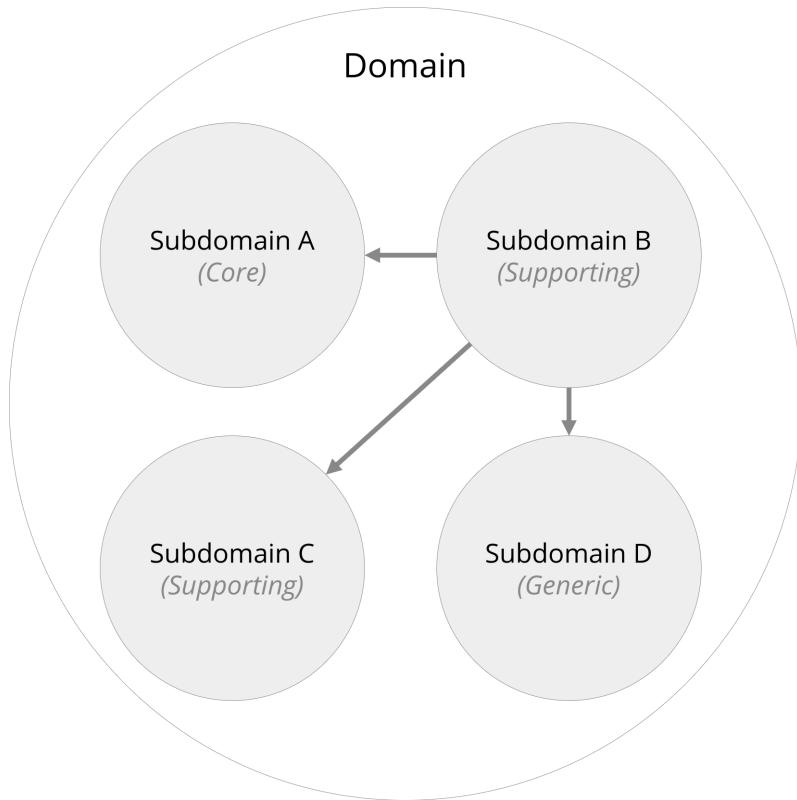


Figure 1.4: Generic Domain Overview

Domain names

While it is important to determine a name for each Subdomain, it is not required for the overall knowledge area. Even more, a software Domain can be a specialized collection of individual parts that are unrelated to each other. [Vernon] uses the abstract term “Domain” for the overarching knowledge area in most diagrams. Also, [Vernon, p. 44] states: “It should be pretty obvious to you what your domain is.”. Whether an explicit name is necessary, depends on the project and its members. Judging from personal experience, it is typically not required for the Domain Model, the source code or other artifacts. Therefore, it can be considered optional. However, if an overall Domain is a common area of expertise that has a universal name, it should be re-used.



Importance of Domain identifications

The Domain of a software and its Subdomains are not expressed as source code. However, their identification is required for being able to create concise and useful Domain Models, which are expressed as code. For this reason, it is crucial to identify the correct involved knowledge areas early on in any given project.

Sample Application: Domains

This section illustrates the definition of the overall Domain and the identification of the Sub-domains for the Sample Application. As first step, the actual problem to solve is formalized and the necessity for a software-based solution is justified. This is followed by describing the selected solution approach. Next, the software Domain is outlined by determining and classifying the contained Subdomains. For this purpose, the main problem statement is analyzed and relevant aspects are processed. The resulting composition is described and visualized. In a real world project, this working step involves human interaction and extensive thought processing. As this book is more focused on the implementation parts, any decorative content to illustrate such a process is avoided.

Problem definition

The first step is to clarify what the actual problem is and whether it can and should be solved with software. For many projects, it is not uncommon to start the development without having these basic things defined. There are various possible reasons for this circumstance. Domain Experts and customers may not be able to express their knowledge and requirements adequately. Also, developers may not understand a Domain well enough, or they might favor technical challenges over building a useful solution. Furthermore, development efforts may start prematurely due to underestimating the complexity of a Domain. Starting with a clear and concise description of the main problem helps to avoid such issues.

1
2
3

Software project checklist

1. Define what the actual problem is
2. Justify the necessity of software
3. Optionally, verify the feasibility
4. Decide on a useful solution approach

Consider the following problem statement: **Distributed teams have difficulties coordinating their work on common projects.** The term “distributed” indicates that members of a team have to be able to work from different locations. This constraint can serve as justification for the need of software. The reason is that analogous solutions cannot really fulfill this requirement. In contrast, software-based solutions can solve the problem easily. The fact that there are already many existing software products that achieve this proves the overall feasibility. With the necessity of software and the feasibility clarified, the next step is to decide on a fitting solution approach.

Solution approach

Before being able to solve a problem, a solution approach needs to be discovered and selected. This can be done by analyzing the problem, gathering ideas, interviewing customers, developing prototypes, testing them and so forth. While all these activities are helpful for finding a good solution, their illustration is not of relevance for this book. For brevity reasons, it is simply assumed that the ideal approach is the implementation of a **Task Board**. Such a software often makes an appearance as a part of a process model like Scrum or Kanban. Therefore, the concept should be familiar to most of the readers. In order to not bloat the complexity, the Sample Application remains free of any such surrounding methodologies.

To do	Doing	Done
Create Domain Model Assignee: unassigned	Identify Subdomains Assignee: John Doe	Decide on solution approach Assignee: Jane Doe
Implement Domain Model Assignee: unassigned		

Figure 1.5: Task Board example

Domain identification

The final step is to define the overall Domain together with its contained Subdomains and their types. The Core Domain for the Sample Application is called **Project Management**. This knowledge area incorporates everything that evolves around projects, teams, coordination of work and, more specifically, also task boards. Excelling in this part bears the possibility to make the software distinctive and successful. The fact that project members can work from anywhere requires to deal with topics such as identity, authentication and authorization. These aspects are contained in the Generic Subdomain **Identity**. Other than the core part, it is free of any problem-specific knowledge. While there are no obvious connections between both parts at this point, it is safe to assume they somehow interact.

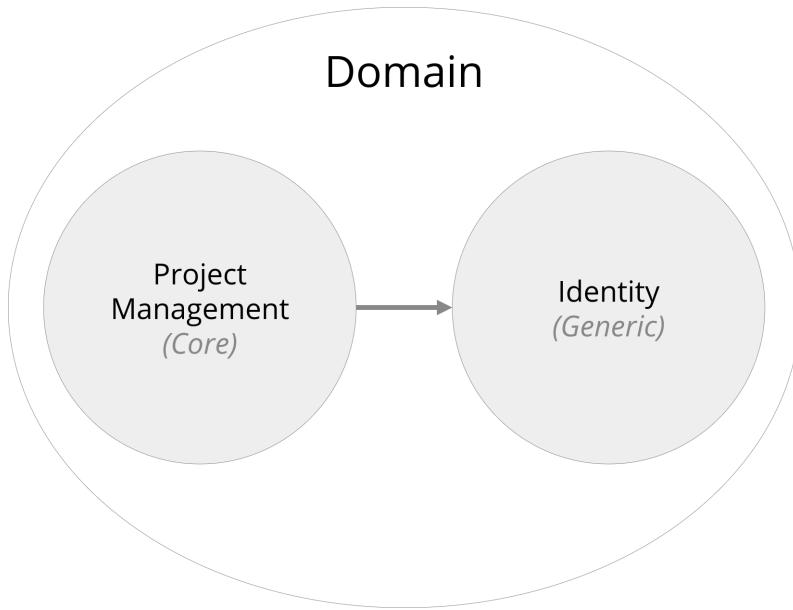


Figure 1.6: Sample Application Domains



What about Supporting Subdomains?

As explained earlier in this chapter, both Supporting and Generic Subdomains are optional parts of a software. In fact, earlier versions of this book defined a Supporting Subdomain for the Sample Application. However, during a refinement of the content, this ill-suited knowledge area was removed.

The problem statement and the solution approach lay the foundation for the Sample Application software. Furthermore, the definition of the overall Domain and its Subdomains serve as first important artifacts. The next step is to create adequate Domain Models for the individual software parts.

Chapter 2: Domain Models

A **Domain Model** is a set of knowledge abstractions that is focused on aspects relevant to solving specific problems. [Evans, p. 3] describes it as “selectively simplified and consciously structured form of knowledge”. The term “model” must not be confused with definitions from other software-related concepts. For example, the pattern Model-View-Controller (MVC) uses it for a technological layer that contains the business logic. Although a Domain Model is typically expressed as code, it is not the code itself or a part of it. Rather, it is structured knowledge that serves as foundation for software and other artifacts. An actual implementation may only reflect a subset of the underlying abstractions, and eventually deals with extraneous technical aspects. Using the explicit term “Domain Model implementation” helps to avoid the ambiguity.



Definition of domain-related terms

Domain

Knowledge area around a problem

Domain Model

Structured abstractions of Domain knowledge

Domain Model implementation

Software solution based on a Domain Model

Structure and components

A Domain Model is a set of abstractions that incorporates both relevant data and behavior. In general, it should not attempt to accurately describe the real world. Rather, a model should consist of targeted abstractions that work best in the context of the respective Domain. Unlike some traditional modeling methods may suggest, the main component is not data that is represented as nouns. Behavior and characteristics are equally important parts. Therefore, verbs, adjectives and even adverbs can be included. The resulting abstractions should neither be too technical, nor so specific that only Domain Experts can understand them. Ideally, every involved project member is able to comprehend the latest Domain Model at each time. Only then it is possible to implement a useful and adequate software solution.

Relation to expert knowledge

The understanding of Domain Experts must not be confused with what the actual Domain Model is. As described in a [section](#) of the previous chapter, the experts are a viable source of specialized information. This makes them an important part of creating and maintaining useful abstractions. They possess complex and partially situational knowledge, which is expressed in some way, often simply through speaking. However, their passed on information needs to be filtered, processed and reduced to the context of the problem space. Also, Domain Experts have to accept the resulting abstractions of a Domain, even if they differ from their perception. This is important, because a Domain Model has to be adopted and consistently applied by everyone in a given project.

Tangibility of abstractions

The idea that a Domain Model is a set of knowledge abstractions may be hard to grasp at first. Trying to think of something without having a concrete tangible representation in mind can be difficult. This is one of the reasons why a model is commonly confused with what are tailored representations of it. While diagrams and documentation are essential, they often serve a specific purpose and do not represent the complete Domain Model. Despite any level of sophistication and detail, almost any expression of information is also a simplification. Even if every relevant aspect could be included unambiguously in one artifact, the result is still an immutable snapshot. Since abstractions of knowledge can change over time, a static representation risks becoming obsolete.

Example: Client project

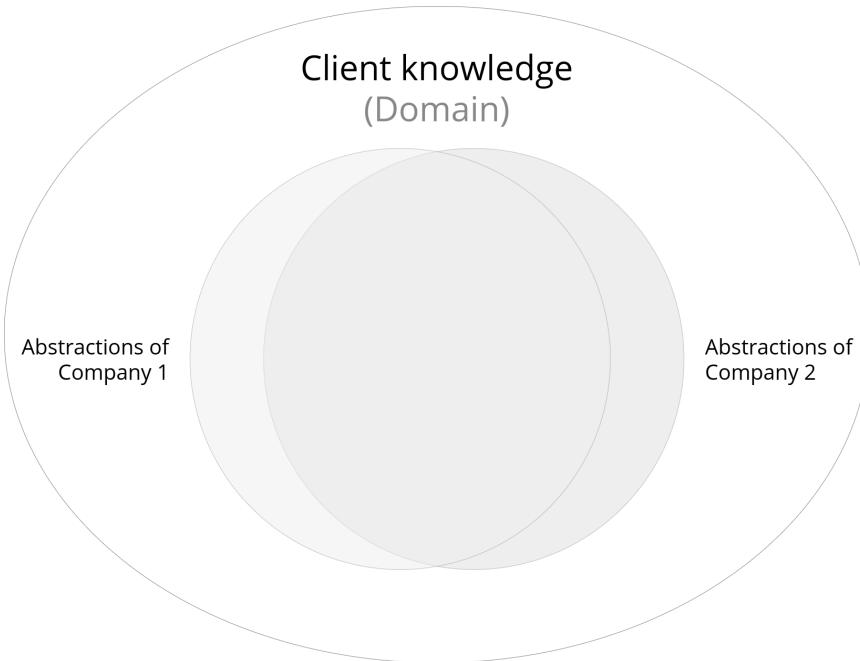


Figure 2.1: Client project Domain

Consider a client who hires a company to build a software. The client explains the initial idea and provides the required Domain knowledge. After defining the problem to solve, the company starts to create fitting knowledge abstractions. While creating documentation during this process can be helpful, the individual artifacts are secondary. The actual Domain Model are the abstractions emerging between the company and the client. Assume that their collaboration runs into disagreements. Another company is hired for building the same software and picks up the created documentation. While the artifacts help to convey the idea of the original model, they do not include every single detail. The new company works out a second set of abstractions, which may even differ slightly from the first one.

Ubiquitous Language

One important aspect for a Domain Model is the language that is used to describe its contents. A **Ubiquitous Language** is a model-based linguistic system for communicating specialized knowledge with high expressiveness and correctness. In this context, the term “ubiquitous”

does not mean that such a language is universal. Rather, it implies its validity everywhere the associated Domain Model is applicable. This especially includes every implementation artifact. The concept promotes not just a specialized form of representation, but rather a linguistic foundation all artifacts should be based on. Furthermore, its consistent application helps deepen the understanding of the Domain Model and revealing new insights. When employing a Ubiquitous Language, it is crucial that every involved member and entity applies it thoroughly.



The “chicken or the egg” dilemma

Developing a Ubiquitous Language seems mutually dependent with the creation of a Domain Model. The language should be tailored to the model, while at the same time all model artifacts should apply the language. However, this is not an actual problem, as the two activities are not meant to be strictly consecutive. Rather, they are best executed in parallel.

Anatomy and documentation

For a Ubiquitous Language, it makes sense to use English as natural language due to its prevalence in software projects. Also, nearly every programming language is built upon it. The vocabulary to create should not only consist of nouns and verbs. Adjectives and even adverbs can be equally important components. Although the documentation of such a vocabulary is not required, it can be very helpful, especially in larger project teams. One common approach is to gather all relevant terms in a simple glossary. However, it must be clear that such an artifact has a temporary character. It is only valid as long as the underlying Domain Model does not change. Otherwise, it needs to be updated in order to reflect the latest insights.

Exemplary glossary for DDD terminology

Term	Meaning
Domain	Knowledge area around a problem
Domain Modeling	Creating specific abstractions of Domain knowledge
Ubiquitous Language	Linguistic system valid in the context of a model

Translations costs

The purpose of a Ubiquitous Language is to eliminate the need for translation and to make communication precise and effective. That is between team members, between different model artifacts and between expert knowledge and source code. Translation is an activity that imposes costs for the involved parties and risks the distortion of facts. Even worse, the inability to translate leads to a total loss of information. When developers do not understand Domain Experts, important aspects may not be reflected in the code. On the other hand, a software expressing model knowledge with development-specific terminology cannot be used correctly by non-developers. Communication problems often lead to the creation of multiple Domain Models that drift apart. An adequate Ubiquitous Language helps to prevent these kinds of issues.

Language example: Application Service

Some time ago, I worked for a company that builds a software for digitalizing processes related to Human Resources (HR). Within the HR Domain, the term “application” refers to the intention of a person to work at a specific job. At the same time, there are software architecture patterns that use the term “application” for a specific technological area. While working on the HR software, I encountered a class with the name “ApplicationService”. In this context, both mentioned meanings of the term would have been conceivable. However, without the HR-specific meaning, the component name seemed awkward, as it redundantly mentioned the architectural layer it belonged to. Only after reading through its implementation, I realized that the name in fact expressed a domain-specific purpose.

Language example: Shared design language

Many years ago, I worked for a company that planned to rework its visual design and to refactor associated implementations. Different product teams developed individual parts of a single platform. For every new feature, a design department created the required layouts. There were visual guidelines, but not all designs were entirely based on them. Also, the guidelines were mostly unknown to the development teams. Only the final layout screens were handed over. While these artifacts contained all visual details, they did not communicate the conceptual model and its components. This caused the codebase to contain a lot of duplication. Creating a shared language for the visual design solved this problem. The consistent use throughout teams and artifacts reduced translation costs and eventually eliminated code duplication.

Domain Modeling

Domain Modeling is the act of creating and refining a Domain Model out of unstructured Domain knowledge. Ideally, this process is executed in an iterative and continuous way. In most cases, it is virtually impossible to transform all relevant aspects into useful abstractions in one step. Any deep understanding of a Domain and the problem space typically only evolves over time. The consequential recurring insights demand to adjust abstractions continuously. Therefore, a Domain Model is best understood as ever-evolving. In fact, insights may happen during the implementation and lead to code adjustments. Naturally, iterative Domain Modeling is a good fit for agile software development. At the same time, it can also be embedded into more traditional processes, though it may be less straight forward.

Knowledge transformation

The transformation of knowledge into suitable abstractions can be done in different ways. There are countless approaches, each with their own characteristics. One example is Event Storming, which focuses on Domain Events and works well with DDD. However, to some extent, it is secondary how exactly the knowledge is processed, as long as it is processed somehow. Consider the previous client example. The client explains their idea, while a company employee creates an informal drawing with shapes and arrows. This artifact is likely to serve two purposes. For one, it is a specific way of capturing information. Secondly, it establishes a Domain Modeling process, as the captured information is abstracted and structured. The client may even actively collaborate in the role of a Domain Expert.

Process integration

Domain Modeling should be integrated into the existing software development process. Typically, there is one initial iteration and recurring follow-up sessions. The initial step can be set up as informal meeting with selected project members. For this part, the participation of Domain Experts and developers is strongly advised. While the experts possess crucial knowledge, the developers are the primary consumers of the resulting Domain Model. The experts can explain their understanding of the Domain. Questions should be brought up so that relevant information is not only conveyed, but also discussed. Eventually, insights occur and relevant facts are revealed. The most important parts must be documented, such as by creating visualizations. On top of the initial modeling, refinement sessions can be made a continuous team effort.

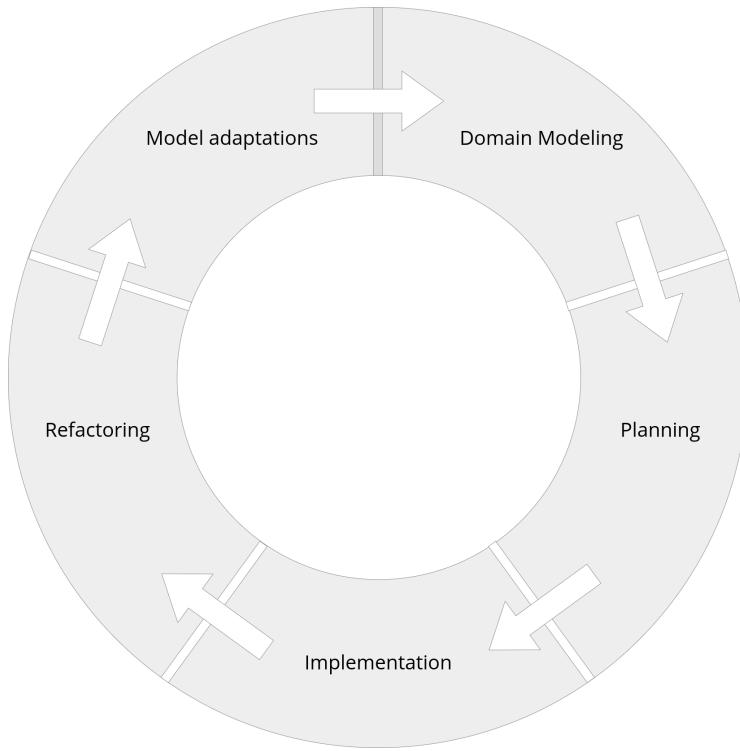


Figure 2.2: Software process with Domain Modeling

Model representations

There are numerous approaches for representing a Domain Model, each with its individual advantages and disadvantages. Although language is an important aspect, it is not mandatory that Domain Models manifest as pure textual representations. Writing notes or simple sentences using the respective Ubiquitous Language enables to express information precisely. One disadvantage is that it bears the risk of getting too detailed. Drawings and diagrams suffer less from too many details, as they are naturally more abstract. However, any associated formalism can introduce overhead. Another option is to create source code. The resulting artifacts are highly tangible representations. One issue is that the code can be mistaken for the actual Domain Model implementation. As with most model-related artifacts, it is important to recognize the temporal validity.

Comparison of Domain Model representations

Approach	Advantages	Disadvantages
Written text	<ul style="list-style-type: none"> - high expressiveness - no special skills required 	<ul style="list-style-type: none"> - some facts may be complex to describe - risk of getting too detailed
Diagrams	<ul style="list-style-type: none"> - naturally more abstract - cheap to create (if informal) 	<ul style="list-style-type: none"> - associated formalism can add overhead - more expensive to create
Code experiments	<ul style="list-style-type: none"> - appealing to developers - tangible representation 	<ul style="list-style-type: none"> - most expensive to create - may be mistaken for implementation

The described methods can be customized and mixed with each other. There are numerous further approaches, which are not explicitly mentioned at this point.



Formalism of diagrams

Formalism in artifacts risks causing an overhead that is contrary to an agile Domain Modeling process. Specifically, unnecessary formalism in diagrams should be avoided when possible. This book uses informal drawings, similar to how it is done in related literature, such as [Vernon] or [Evans]. This way, the reader is not required to be experienced with specific diagram types.

Complementary representations

The various approaches of expressing Domain Model knowledge inherently differ in what information they convey and how this is done. Many times, the explanation of behavior and characteristics works better with written text. On the other hand, drawings can make it easier to comprehend a larger system and the relations of its parts. The previously mentioned options for model representations are not meant to be interchangeable, as each of them serves a specific purpose. Rather, they are complementing each other, and their combination allows to gain a more comprehensive understanding of a Domain Model. This is one of the reasons why it can be helpful to use different ways to convey knowledge. Multiple expressions can produce distinctive but complementary viewpoints.

Sample Application: Domain Models

This section describes the creation of the Domain Models for the Sample Application. The activity depends on the results from the [sample section](#) of the previous chapter. The problem

definition, the solution approach and the identified knowledge areas establish the required foundation. They enable to understand which Domain Models must be created and what purpose they must serve. There are multiple steps involved in the modeling activity. First, all relevant Domain knowledge is gathered and filtered. Then, the resulting information is transformed into cohesive sets of useful abstractions. As next step, the Domain Models are expressed through different exemplary representations. Besides their main purpose, these artifacts also illustrate the underlying Ubiquitous Language. Finally, the linguistic system is presented through a simplified glossary.

Domain knowledge gathering

The following lists are collections of relevant Domain knowledge, which are grouped by their respective Subdomain:

1. Project Management knowledge:

- Tasks represent work that needs to be done
- Team members are responsible for tasks and work on them
- When a task's work is completed, it should be clear to the team
- More complex tasks require detailed descriptions
- Team members may interrupt their work and hand it over
- Teams must know which tasks are open, which are worked on and which are finished
- Tasks can become obsolete throughout a project
- Also, new tasks may come up at any time
- For each project, there is one responsible person
- Teams use a task-based working approach
- A team consists of people with individual roles
- Team members have to be able to work from anywhere
- Teams normally consist of five to ten people
- Team members can leave a project at any time
- New team members may join a project team

2. Identity knowledge:

- The same person can work on multiple projects
- A person uses an e-mail address for identification
- E-mail addresses of persons can be changed
- Every person uses a custom nickname
- Administrative access to the software must be possible

Domain Modeling

With the relevant Domain knowledge in place, the initial Domain Modeling activity can be performed. The first step is to analyze each statement and isolate its relevant aspects. Then, the information is distilled and transformed into structured knowledge. The resulting abstractions become part of the respective Domain Model. For example, while none of the statements express it directly, some imply that tasks have three possible states. This can be used to derive the fact that tasks on a board are divided into three sets. In contrast, other statements may also contain irrelevant information. While the average team size is a domain-related knowledge item, it is not of further relevance. The complete set of Domain Model statements is presented as items grouped together by their Subdomain:

1. Project Management Domain Model knowledge:

- Tasks on a board are split up into three status groups
- The status groups are named “Todo”, “In Progress” and “Done”
- New tasks can be created and existing tasks can be deleted
- Tasks always have a title and optionally a description
- Tasks have an assignee, but can also be unassigned
- The status of a task can be updated at any time
- When a task is being worked on, it must be assigned
- An assignee of a task can be removed or changed
- A project is a combination of a team and a task board
- Each project has an owner, which may also be a team member
- Projects always use a single task board
- Projects must be distinguishable from one another
- Project teams are specific to one individual project
- Teams can have an arbitrary number of members
- New members can join a team and existing ones can leave
- Upon leaving, any assigned tasks must become unassigned
- Each member has one specific role
- Roles are profession-related specializations
- A team member is linked to a single user account

2. Identity Domain Model knowledge:

- People can authenticate themselves

- A person owns a single user account
- User accounts are linked to unique e-mail addresses
- E-Mail addresses can be changed at any time
- User accounts contain a username chosen by the owning person
- Each user has one specific role
- Roles determine what kind of access a user has

These lists are the resulting artifacts of an initial Domain Modeling activity. Consequently, they are also a communication tool to convey the idea of the emerging Domain Models. The diction of the statements may suggest that they are software requirements. This is not the case. Rather, they are an informal collection of rules and characteristics. As previously explained, a model itself is structured knowledge and not one specific representation. The same applies here. However, without working together as an actual team, it is hard to share the exact same knowledge. For this reason, the lists serve as surrogates for the Domain Models of the Sample Application. Since written statements are not the only useful representation type, the next section provides two exemplary types of visualizations.

Visual representations

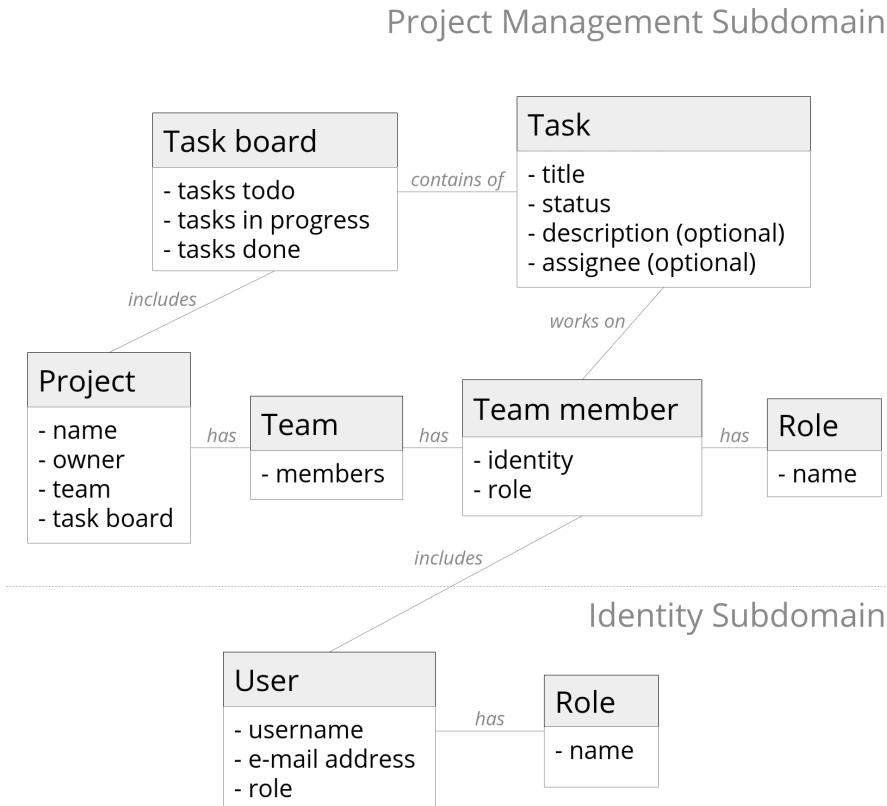


Figure 2.3: Sample Application: Domain Model components

The first diagram represents an alternative expression of the previously introduced Domain Model knowledge. As mentioned earlier, the visualization is informal and only remotely resembles the characteristics of a class diagram. In this case, the boxes represent meaningful and self-contained concepts of the model, and their contents describe enclosed information. Lines between the boxes indicate relationships, which are further classified by their respective text label. Overall, the drawing is divided into the two identified Subdomains. This kind of visualization is useful for illustrating components and their relationships. However, it is only one exemplary approach to express Domain Model knowledge. There are also many other options. For example, when aiming to emphasize behavior, one possibility is to identify relevant events, as illustrated in the second visualization.

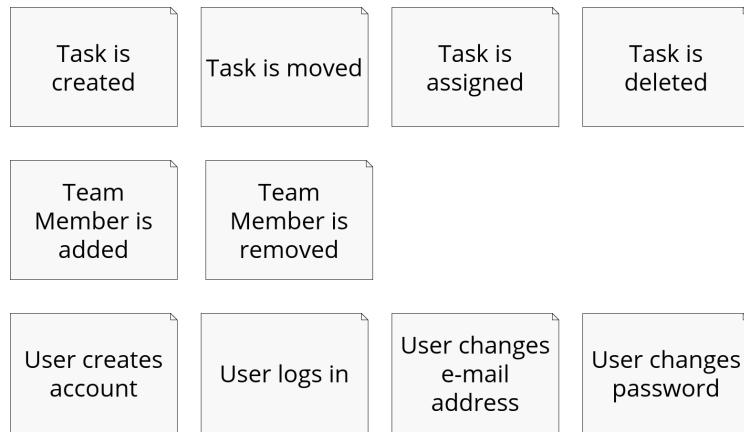


Figure 2.4: Sample Application: Domain Events ideation



Refinements of the Domain Models

As previously explained, a Domain Model constantly evolves. Consequently, the illustrated abstractions in this section are to be seen as initial draft. Specifically, the collection of events does not represent a complete set of Domain Events. Throughout the following chapters, there are domain-specific insights as well as technological circumstances that cause the abstractions to change.

Language and glossary

The Ubiquitous Language for the Sample Application is illustrated with a simplified glossary. However, it is not defined in full detail. There are multiple pragmatic reasons for this. One is that the previously shown artifacts already contain implicit term definitions and partially illustrate the underlying language. Secondly, a glossary is first and foremost a tool for establishing a common vocabulary in a project team. The third reason is that the Domain Model applies terminology that conforms to industry standards. Therefore, the reader is expected to be experienced with the vocabulary. Despite the incomplete language definition, the application of the Ubiquitous Language is essential nonetheless, especially for the Domain Model implementation.

Simplified glossary for the Sample Application

Subdomain	Term	Meaning
Project Management	Task board	Structured collection of related tasks
Project Management	Task	Working item
Project Management	Assignee	Responsible person for a task
Project Management	Project	Purpose-driven combination of team and task board
Project Management	Owner	Responsible person for a project
Project Management	Team	Collection of members
Project Management	Member	Person as part of a team
Project Management	Role	Profession-related specialization
Identity	User account	Collection of person-specific information
Identity	Username	Alias for a person that is linked to an account
Identity	Role	Description of global access capabilities

The statement collections, the visualization and the exemplary glossary represent the means to convey the idea of the Domain Models. Their combination provides the necessary setup for being able to define useful conceptual boundaries and potential technological units.

Chapter 3: Bounded Contexts

A **Bounded Context** is a conceptual area that determines the applicability of a Domain Model. [Vernon, p. 62] describes it as “explicit boundary within which a Domain Model exists”. [Evans, p. 384] defines it as “intention to unify a model within certain boundaries”. This implies that a particular set of abstractions must be interpreted in only one way within a defined scope. Doing so enables the use of the associated Ubiquitous Language without the risk of misunderstandings or the need for translation. There are many situations where Bounded Contexts are utilized for technical decisions and sometimes even mistaken for Software Architecture. While conceptual boundaries and software structures commonly align with each other, the primary goal is to define Domain Model applicability.

Relation to Domains

The concepts Domains, Domain Models and Bounded Contexts stand in close relation and are commonly confused with each other. Overall, they are the more controversial and discussed parts of DDD. One possible reason is that, unlike with technical topics, it is more difficult to differentiate the theoretical parts. The **Domain** around a problem and its Subdomains are areas of related knowledge. This means that they categorize existing information. In contrast, **Domain Models** are individual abstractions using a custom language that are created out of existing knowledge. Their contents and their structure are not predetermined and can be specific to a software. The same applies to **Bounded Contexts**. They are individual conceptual boundaries with custom names that define the applicability of their enclosed Domain Models.



Differentiation between terms

Domain

Collection of existing knowledge

Domain Model

Individually created abstractions

Bounded Context

Individually defined boundary

Model and context sizes

The ideal sizes for Domain Models and Bounded Contexts depend on the associated problem space and the involved Subdomains. Other than naturally given knowledge areas, the individual abstractions and their conceptual boundaries can be structured freely. For Domains with a low complexity, it can be advantageous to have one large Domain Model inside a single context. At the same time, smaller models in separate conceptual boundaries can have various benefits. While general recommendations exist, there are no strict rules for the sizes of Domain Models and Bounded Contexts. The following subsections describe different approaches together with their characteristics and implications.

Large Domain Model

One option is to create a single large Domain Model for a complete Domain. The goal is to have the exact same interpretation across organizational units, individual people and artifacts. In the best case, this approach provides uniformity of knowledge and eliminates ambiguity. While these are promising effects, there are various risks. The size and the scope of a Domain Model are typically proportional to its complexity. This means that larger models are harder to comprehend. Also, with more information contained, more reasons exist why a model may need adjustment. Furthermore, when many people share the same interpretation, they all need to be informed about every single change. Independent of the motivation for a large Domain Model, this approach should be considered with caution.

Single unified interpretation

The most extreme variant of a large Domain Model would be to create universal abstractions for all existing human knowledge. This would result in a single unified and all-including view of every piece of information. Anything could be explained and discussed by using this one interpretation. Also, it would free from the need of abstracting knowledge in order to create purpose-specific individual Domain Models. However, such a compound of abstractions would contain too much information and would be too complex to comprehend at large. Furthermore, it would have to be changed constantly with every new insight in any subordinate subject area. Otherwise, it would become obsolete quickly. As appealing as a single all-encompassing Domain Model might seem, it is likely to be not very practical.

Example: Meeting software

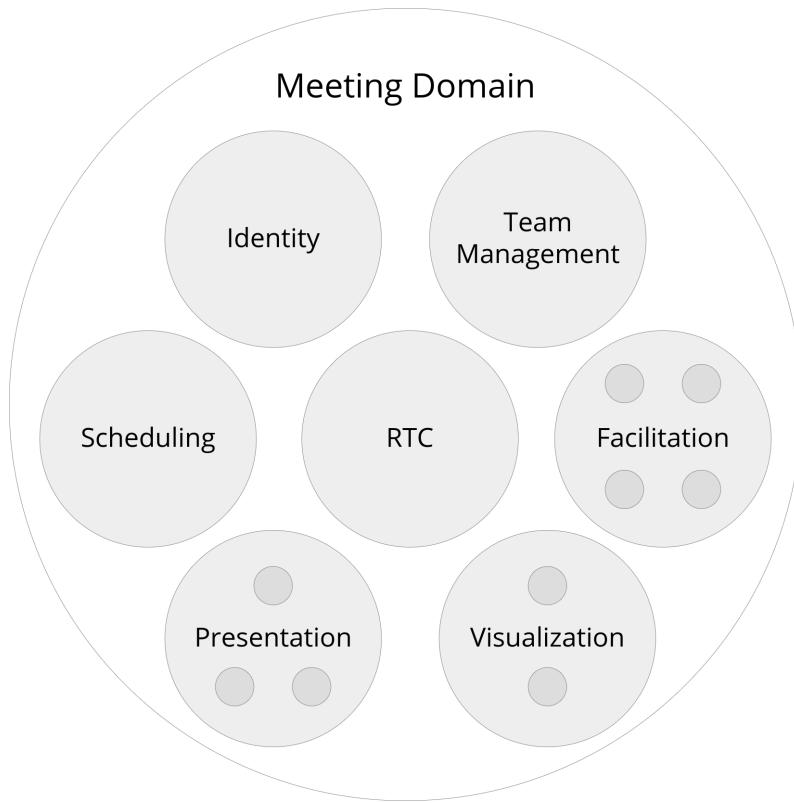


Figure 3.1: Meeting software Domain example

Consider working for a company that develops a software for meetings. The associated overall Domain is broad and complex as it encloses many Subdomains. There are generic topical areas such as Identity, Team Management and Scheduling. Then, there are supporting areas like Real-Time Communication (RTC). And most importantly, there are the three core knowledge areas Facilitation, Presentation and Visualization. Each of them itself again contains multiple Subdomains. Assume that the company decides to abstract all the knowledge into one unified Domain Model inside a single Bounded Context. The main motivation is to use the same interpretation across the whole organization. Despite possible advantages, this is likely to result in a model that gets abandoned quickly. The complexity, inflexibility and maintenance costs would make it practically unusable.

Small Domain Models

Separate small Domain Models enclosed in their own Bounded Context are generally superior to having one large model. This is because smaller ones have a lower complexity, a higher expressiveness and are less subject to change. Even if some parts depend on each other, they do not necessarily have to be combined into one common model. Relationships and integrations can be designed explicitly as connections across Bounded Contexts. Many times, this is more versatile than joining multiple parts together. Also, knowledge areas that are completely unrelated do not gain any advantage of a common model inside one context. Even worse, doing so only adds unnecessary complexity. As with other concepts of Software Development, low coupling and high cohesion are important Domain Model characteristics.

Alignment with Subdomains

There are different possibilities for the alignment of Subdomains and Bounded Contexts. Ideally, a context includes a single Subdomain or a selected part of it. This way, the Domain Models inside a conceptual boundary are only concerned with one specific knowledge area. As a consequence, the models typically have a lower complexity. However, this approach is not always feasible. There are situations where a Domain Model must incorporate knowledge from multiple Subdomains and necessitates a larger conceptual boundary. In such a case, one possibility is to create multiple smaller abstraction sets and define specific relationship types. In general, a one-to-one alignment of Subdomains and Bounded Contexts is desirable, but not strictly mandatory.

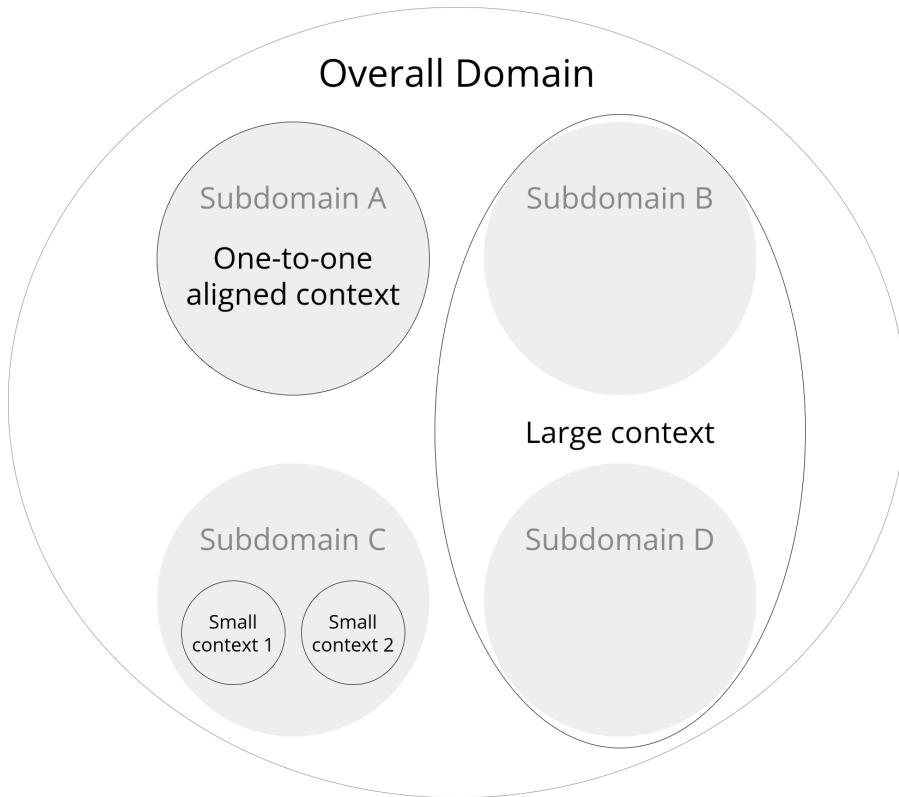


Figure 3.2: Alignment of Subdomains and Bounded Contexts

Contexts and language

Bounded Contexts also determine the area in which a Ubiquitous Language is applicable. This is because every such language is related to a Domain Model, which itself is enclosed in a certain context. This means, the associated terminology can only be used safely inside the respective bounds without risking communication issues. Outside the enclosing context, a Ubiquitous Language may have no meaning. Despite useful conceptual boundaries, there can be situations where identical terms with different meanings exist. Typically, this only happens when multiple Subdomains are abstracted into a common Domain Model. One solution approach is to choose alternative or overly explicit terms. However, this circumstance can be an indication that the conflicting abstractions should be modeled separately in different Bounded Contexts.

Example: Web shop builder

Consider developing a system that enables the creation and hosting of web shops. Customers can build their own platform for selling arbitrary goods. As a consequence, each of them has its own customers. This makes the term “customer” ambiguous. It can stand for the people that operate a shop, or for the ones that buy goods on a platform. One initial modeling approach may lead to a single Domain Model inside one Bounded Context. In this case, the collision of terms can be mitigated by naming the second group of people “web shop customers”. The cleaner and more modular solution is to create separate Domain Models. Most likely, the two knowledge areas around the different customer types have no significant relation to each other anyway.

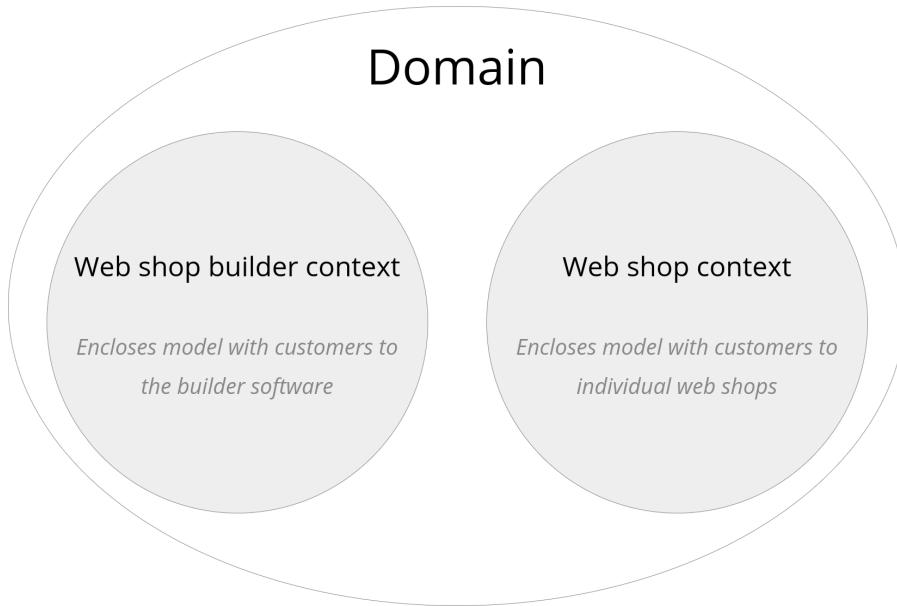


Figure 3.3: Web shop builder Bounded Contexts

Technological boundaries

Bounded Contexts are of conceptual nature and do not necessarily relate to software structures. Their main purpose is to define the partitioning of a Domain into a set of delimited models. Even more, there is no rule that models must be expressed as code at all. The field of Lean Product Development promotes the concept Concierge Minimum Viable Product.

This approach suggests to evaluate the potential of an idea by executing it manually first. As example, consider a recommendation feature for an online job platform. The functionality can be prototyped by hand-picking job recommendations without developing software. Still, the manual work represents an application of a particular Domain Model inside a Bounded Context. This exemplifies the importance of differentiating between conceptual boundaries and software-related aspects.

Exemplary delimiting software mechanisms

Mechanism	Description
Modules	Logical division inside a monolithic software
Processes	Runtime separation of individual executable programs
Infrastructure	Infrastructural distribution of distinct programs

Despite their conceptual nature, it is common and often useful that contexts align with some form of technical division. [Vernon, p. 71] explains that it “doesn’t hurt to think about a Bounded Context in terms of the technical components that house it”. In fact, Domain Models and their boundaries form units that show similar characteristics as delimiting software mechanisms. Therefore, it can make sense to represent their shapes as individual modules, as standalone processes or even as separated infrastructure. The right scale and division mechanism depends on the individual project and the corresponding Domain. However, it is crucial to understand that these exemplary constructs are technological concerns and not the conceptual boundaries themselves.

Relationships and integration

When a Bounded Context depends on another one, it can be helpful to classify the relationship and the integration approach. For this purpose, [Evans] and [Vernon] define multiple organizational and integration patterns. The code examples in this book use a varying combination of three patterns for connecting the implementations of dependent contexts. When one software part must expose access to selected components, it acts as **Open Host Service**. Whenever data from one part is used in a foreign context implementation, an **Anti-Corruption Layer** is put into place. Finally, the use of an event-based message exchange is assumed to constitute a form of **Published Language**. Apart from the brief mentioning of the used concepts, this book is not further concerned with the patterns.



Further details on the patterns

The organizational and integration patterns for Bounded Contexts are covered in full detail by both [\[Evans\]](#) and [\[Vernon\]](#). In general, the patterns are helpful to classify and categorize organizational and technical relationships. However, they are not of further use for the scope of this book.

Context Maps

A **Context Map** is an informal drawing of Bounded Contexts to visualize their relationships and their integrations. Similar to the other drawings in this book, the elements of such a map are typically represented by circular shapes. Each included context should provide its name. Connections between distinct parts are visualized with lines. Their relation and integration is explained through a short textual description. Context Maps can serve multiple purposes at once. On the one hand, they illustrate the structure of conceptual boundaries and their interdependencies. On the other hand, they can declare the technical patterns that are used to integrate the implementations of different contexts. However, when using them to describe concepts that are independent of technical concerns, the integration details may be excluded.

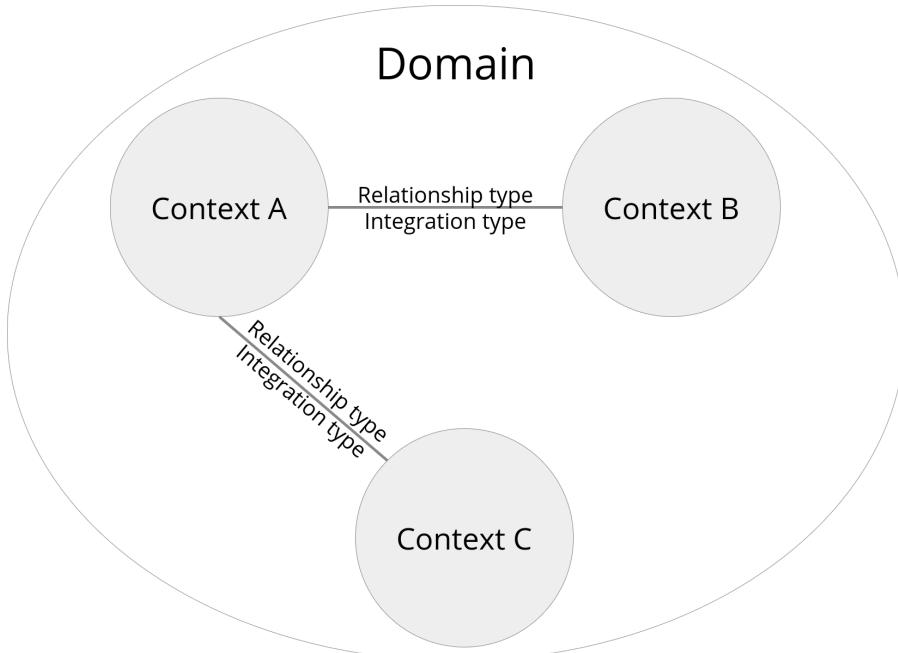


Figure 3.4: Generic Context Map

Sample Application: Bounded Contexts

This section illustrates the process of defining the Bounded Contexts and the subsequent Domain Model partitioning for the Sample Application. The identified Subdomains and the Domain Model knowledge from the last chapters are used to justify the conceptual boundaries. Their definition is complemented with a simplified Context Map to visualize associations and dependencies. However, the connections between contexts only define the exchanged information and its direction without classifying the relationship or integration. This has multiple reasons. For one, there is no organization involved in the Sample Application and therefore there are no organizational relationship types. Secondly, the technical integration part is a technological aspect and only covered in the subsequent chapters. Therefore, the following definitions and the Context Map exclusively describe conceptual boundaries and associations.

Standard approaches

The simplest possibility is to define one large Domain Model inside a single Bounded Context. For the Sample Application, this can be a valid strategy. The overall model has a manageable complexity, it remains largely unchanged over time, and the only collaborator is the reader. Yet, this approach may lead to an overly complex or ambiguous abstraction of the Domain. The reason is that multiple distinct knowledge areas would be combined into one Domain Model. An alternative is to aim for a one-to-one alignment with Subdomains. This requires one context for Project Management and one for Identity, each with a smaller Domain Model. However, simply following the standard recommendation does not take into account what is the most useful set of conceptual boundaries.

Language considerations

Another influencing factor for the definition of Bounded Contexts are the potentially separate languages within different Domain Model subsets. The Project Management part defines the term “role” as a Software Development specialization. In contrast, the Identity part uses the same word to differentiate user types. Due to this collision in terminology, the two parts cannot be simply joined together into a common model. One pragmatic solution is to employ more specific terms, such as “project role” and “user role”. Although this would solve the conflict, the issue hints to a more important aspect. Project Management and Identity are distinct knowledge areas, which even use the same term for different concepts. Consequently, there is little advantage to expect in modeling them together inside the same context.

Component relationships

Relationships between Subdomains or between subsets of a Domain Model are important factors for conceptual boundaries. For the Sample Application, there is only one connection between the identified Subdomains. Every team member of a project references the identity of a user. In this case, the cost of a context-overarching relationship must be compared to the implication of a large Domain Model. Furthermore, inside the Domain Model part for Project Management, multiple subordinate areas can be defined. One option is to introduce two contexts, one with task boards and tasks, and the other one with projects and teams. Again, the approach of maintaining relationships between these potential contexts must be compared to combining them in a single model.

Decision and definition

The definition of Bounded Contexts should be reinforced with project-specific arguments. Simply following guidelines can be counterproductive. Even more, a conscious violation can make sense. For the Sample Application, there are mainly two approaches. One is to create a Bounded Context for Project Management and another one for Identity. This results in a one-to-one alignment with Subdomains. Another approach is to further subdivide Project Management into Domain Models and Bounded Contexts for Task Board and Project. While both abstraction parts belong to the same Subdomain, they only have a few points of contact. Consequently, it seems more useful to maintain explicit relationships instead of modeling them together. In the end, the Sample Application defines the Bounded Contexts **Task Board**, **Project** and **User**.

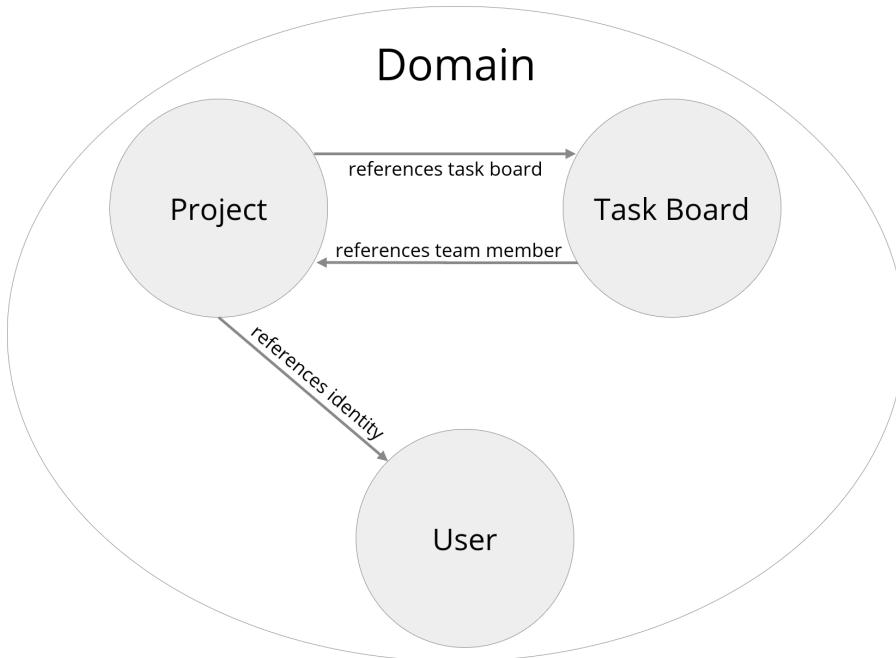


Figure 3.5: Sample Application Context Map



Bounded Context names

As mentioned earlier in this chapter, Bounded Contexts have a custom name. Ideally, each name is concise and intention revealing. Whenever a context is aligned one-to-one with a Subdomain, it can make sense to re-use its name. Also, it is also acceptable to include corporate or even brand terms.

The Bounded Context definitions determine the conceptual boundaries and the set of individual Domain Models for the Sample Application. On top of that, they lay the foundation for starting to work on the actual technological aspects.

Chapter 4: Software Architecture

The **Software Architecture** defines the structure of a software as well as the interaction possibilities between its contained parts. There are multiple commonly applied architectural patterns that fit especially well with DDD. Generally, all of them aim to build software in a modular and decoupled way with a clean separation of concerns. In fact, many of the approaches share the same fundamental ideas, concepts and terminology. Dividing software into multiple architectural parts can happen on different levels. The activity can target the design of a complete system or it can be about selected areas. For software that applies DDD, it typically affects either the overall structure or individual implementations of specific Bounded Contexts. Amongst other things, this means that different subsystems can have different architectures.



Occurrences of unfamiliar concepts

Some parts of this chapter refer to concepts that are only covered at a later point. As explained in the preface, the book's chapter order aims to avoid such a situation. However, this circumstance is inevitable when aiming to provide an upfront introduction of common software parts and architectural patterns.

Common software parts

Every software with a certain degree of complexity can be divided into multiple parts with distinct roles, responsibilities and relationships. Although the two architectural patterns explained in this chapter differ in some aspects, their overall layout is aligned with each other. For this reason, it makes sense to first describe the most common individual software parts independent of the specific patterns. Typically, a software that applies DDD can be divided into **Domain**, **Infrastructure**, **Application** and **User Interface** (UI). Each of the previously introduced parts is described individually in the following subsections. Note that this partitioning is again first and foremost of conceptual nature. Therefore, it does not automatically imply any specific technological division, such as separate directories or software modules.

Parts of a software

Software Part	Responsibility
Domain	Implementation of the Domain Model
Infrastructure	Technical functionalities with optional external dependencies
Application	Use case execution and management of transactions and security
UI	Interface for humans or machines to interact with the software

Domain

The Domain part hosts the Domain Model implementation. This typically includes Value Objects, Entities, Domain Services, Invariants, Domain Events, Aggregates, Factories and possibly persistence-related abstractions. There should be as few outbound dependencies as possible. Unlike the abstractions, the concrete persistence implementations belong in the Infrastructure part, as they contain technological details. The Domain Model implementation is the heart of every software and therefore its most important part. This is the case for both the overall knowledge area and for individual Subdomains. Within each conceptual boundary, the corresponding Domain part is always of the highest importance. For example, consider an authentication service as part of a larger system. Despite its generic nature, its Domain Model implementation is still more important than domain-agnostic functionalities of other parts.

Infrastructure

The Infrastructure part must contain all technical functionalities that neither belong to the Domain nor to the Application. [Vernon, p. 122] explains that this includes “all the technical components and frameworks that provide low-level services for the application”. Although it is not the primary intent, the contained elements may have external dependencies and make use of specific technologies. Typically, this part houses components for persistence and information-exchange-related mechanisms. Furthermore, it commonly contains low-level functionalities such as identifier generation, password encryption or validation utilities. One typical example for a persistence mechanism is the implementation of a Repository. While the corresponding interface conceptually belongs to the Domain part, the concrete implementation is an infrastructural concern.

Application

The Application part is responsible for the execution of domain-specific use cases. For this purpose, it utilizes infrastructure functionalities and works with Domain components. This means, it is a direct consumer of the previously described parts. Typically, a use case execution incorporates loading persisted information, executing domain-specific behavior and eventually saving changes. Also, it can include working with multiple components and orchestrating their interaction. Conceptually, this software part acts as a facade around a Domain Model implementation. Therefore, it must also deal with input validation. [Vernon, p. 120] explains that it is further responsible for “persistence transactions and security”, for which the Infrastructure provides required functionalities. According to [Vernon, p. 68], “user interface components and service-oriented endpoints” are its direct consumers.



Ambiguity of the term Application

Using the term “Application” as name for an architectural part of a software can be confusing. This is because it is also used for self-contained executable programs. However, in the context of Software Architecture the term rarely stands on its own. Most commonly, the software part is referred to as “Application Layer” and its components are implemented as “Application Services”.

User Interface

The User Interface part contains functionalities that are required for a human or a machine to interact with a software. This involves displaying information, accepting and translating inputs, communicating with Application Layer components and presenting returned results. Furthermore, it can also incorporate client-side validation. While this architectural part is allowed to consume domain-related functionalities, it must not contain them directly. For web-based software, the most common approach is to create a browser interface using HTML, CSS and JavaScript. However, the User Interface is not restricted to artifacts that are interpreted on a client. For example, when applying an MVC-based architecture with server-side controllers, their implementations also belong to this architectural part. Conceptually speaking, it consists of everything in between the clients and the Application components.

Layered Architecture

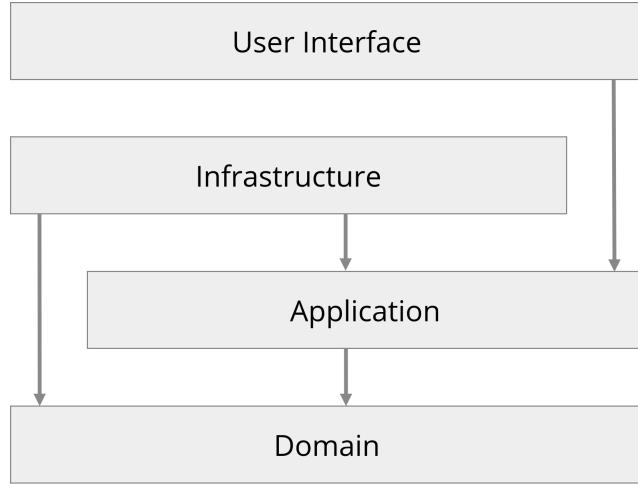


Figure 4.1: Layered Architecture

The **Layered Architecture** divides software into horizontal layers, of which each may only depend on itself and below. In general, this approach does not dictate the number of layers or their responsibility. However, for DDD-based software, they typically consist of the four previously described parts. The User Interface is at the top and depends on the Application. Next is the Infrastructure, which depends on both the Domain and the Application. The third layer is the Application, which depends on the Domain. Finally, the Domain sits at the bottom. Normally, both the Application and the Domain would depend on the Infrastructure. This dependency can be inverted through the use of abstractions. The Domain and the Application define and consume interfaces, for which the Infrastructure provides implementations.

Onion Architecture

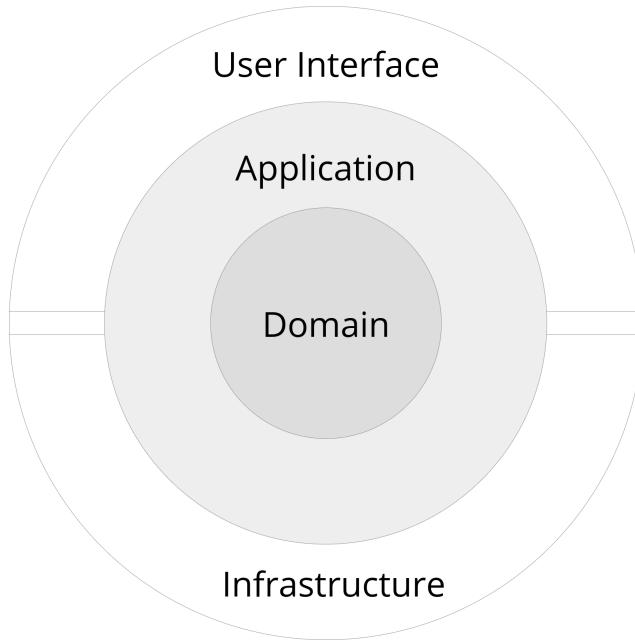


Figure 4.2: Onion Architecture

The Onion Architecture also divides software into layers with the rule that dependencies can only go inwards. The pattern is named after its visual representation, which resembles the shape of an onion. There are two noteworthy differences to the Layered Architecture. For one, the Onion Architecture is more specific to DDD and typically defines three or four layers. Secondly, a layer can contain multiple parts. The User Interface and the Infrastructure are placed on the same area. Both of them depend on the Application part in the next layer. Again, the dependency direction to Infrastructure is inverted through the use of abstractions. The Application depends on the Domain, which sits at the core. Overall, the Onion Architecture is very similar to the Layered Architecture.

Approach for this book

This book and its implementations follow the shared approach of the previously described Layered Architecture and the Onion Architecture. However, there is no inversion of dependencies through the use of explicit abstractions. Specifically, there are no interfaces for

infrastructural functionalities. Rather, the underlying concept is used to derive three code design rules that produce an equivalent architectural layout. First, the Domain part must not depend on the Infrastructure layer in any way. Secondly, the Application Layer may use infrastructural implementations, as long as they are provided from the outside at runtime. Third, all Infrastructure components that are consumed by other layers must have abstract signatures that are free of technological details. In sum, these rules help to achieve similar practical advantages as when using interfaces.



What about other patterns?

There are other similar architectural patterns that also fit well with DDD. Ports and Adapters focuses on separating a software core from different types of inbound and outbound communication. The Clean Architecture is a combination of multiple approaches. Many times, the practical differences between all those patterns is minimal. One could even say, everything is some kind of Layered Architecture.

Sample Application: Software architecture

This section defines the initial Software Architecture for the Sample Application and its subordinate parts. The artifacts from the first three chapters serve as conceptual foundation for creating a meaningful architecture. The goal is to provide an initial draft of a useful software layout. While the result outlines a conceptual division and the interaction between individual parts, it does not dictate a specific implementation style. The patterns on how to express individual aspects as code are only covered in subsequent chapters. Also, the illustrated architecture does not prohibit to apply additional useful patterns. Specifically, the later use of Event-driven Architecture, CQRS and Event Sourcing must not be seen as replacements, but rather as extensions.

High-level structure

There are two useful approaches for the architecture of the Sample Application. One is to define a single unit for the implementation of all three Bounded Contexts. Effectively, this produces a monolithic software. The alternative is to define a separate area for each context implementation. This approach favors the separation of the conceptual boundaries and their enclosed Domain Models. Furthermore, it is useful when aiming to separate the implementations, be it logically or as individual programs. Therefore, the Sample Application

architecture defines the three areas **Project**, **Task Board** and **User**. Each of them is concerned with the implementation of their associated Bounded Context. Internally, they apply the Onion Architecture pattern. This is complemented with one shared area for infrastructural functionalities and domain-related utilities.

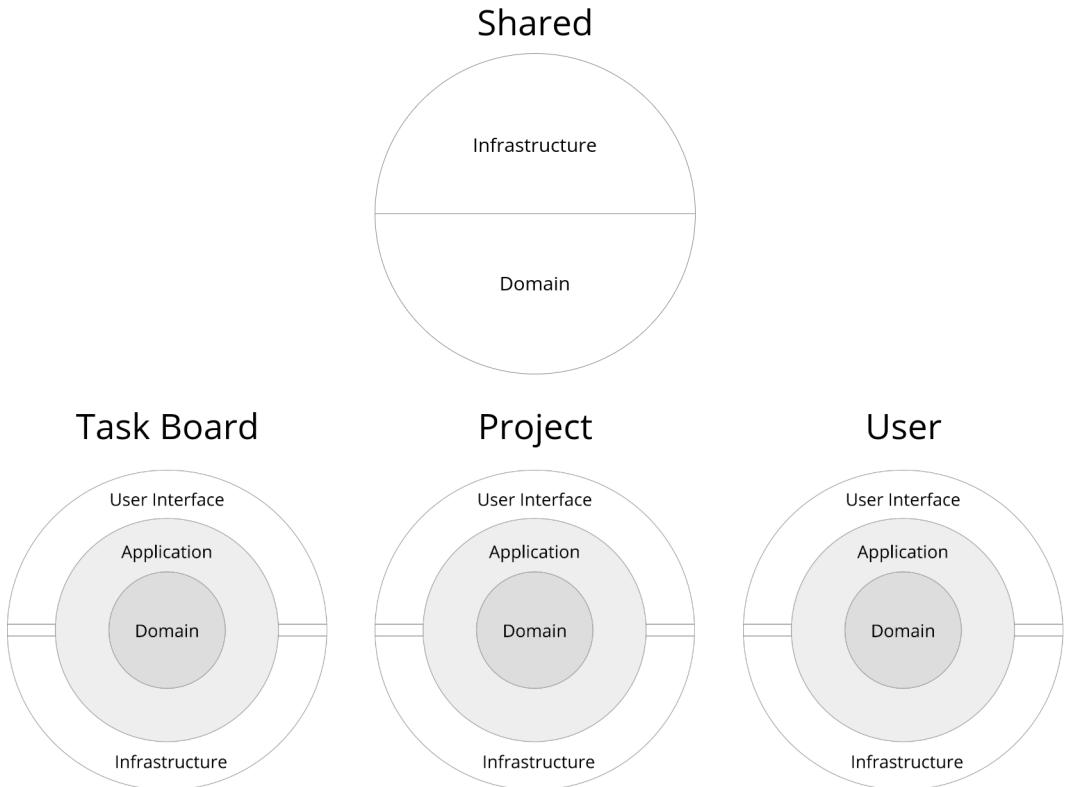


Figure 4.3: Sample Application architecture

Logical division

The introduction of four architectural areas, of which three align with Bounded Contexts, allows to derive a useful technological division. At a minimum, the implementations of the different parts should be separated in some way. This can be achieved by placing them in different physical locations, such as individual subdirectories. For this purpose, the directories `project`, `task-board` and `user` are defined to host the implementations for the respective Bounded Contexts. Within each of them, the parts of the Onion Architecture are reflected as subdirectories. For shared functionalities, the directory `shared` is defined,

which is further subdivided into `domain` and `infrastructure`. As first approach, the software is assumed to operate as single process. Consequently, any integration between different areas can be done via direct code execution.

The following snippet outlines the Sample Application directory structure:

Sample Application: Directory layout

```
/shared
  /domain
  /infrastructure
/project
  /application
  /domain
  /infrastructure
  /ui
/task-board
  /application
  /domain
  /infrastructure
  /ui
/user
  /application
  /domain
  /infrastructure
  /ui
```

The Software Architecture and the directory layout provide the required knowledge that allows to start with the actual implementation. Before focusing on the patterns on how to express a Domain Model as code, selected code quality aspects are discussed.

Chapter 5: Code quality

For the majority of software projects, the source code is the most important artifact. Besides containing the instructions for the actual executable programs, the code also represents an expression of the Domain Model. This expression can ideally be understood not only by developers, but also by other technology-affine project members. In general, it makes sense to aim for a high code quality. For that, there are essential supporting aspects such as readability and coherence. Furthermore, there are design principles that also contribute to this goal. This chapter explains selected aspects and principles that enable to achieve a high code quality. While the topics are partially explained with regard to Domain Model implementations, they are equally relevant for other software parts.



Optional Chapter

The code quality aspects and principles covered in this chapter influence most of the implementations in this book. If you are familiar with them, feel free to skip this chapter. Also, if you disagree with some of the ideas, the remaining book content remains valuable.

Model binding

A Domain Model and its implementation must be bound strongly to each other. Artifacts, such as drawings or text, are useful for Domain Modeling and documentation. The implementation can serve the same purposes, but also defines the actual executable programs. While it may be acceptable that other artifacts become temporarily outdated, the software must always reflect the current Domain Model. Whenever this is not the case, all affected code parts must be adjusted. As explained in the [second chapter](#), code experiments can blur the lines between abstractions and their implementation. The therein contained statement also applies in this context: The source code is not the same as the Domain Model. Nevertheless, it is a crucial, contemporary and ideally synchronized type of artifact.

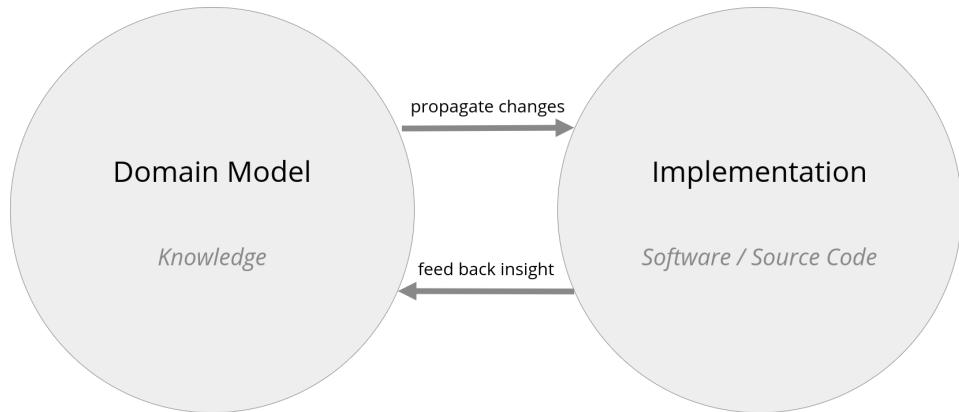


Figure 4.1: Model binding

Every change to a Domain Model requires to adjust the associated implementations. At the same time, when software modifications reveal new insights, the underlying Domain Model evolves implicitly. Apart from the source code, it can also be important to continuously update other relevant artifacts. Not doing so causes multiple different Domain Models to emerge, eventually leading to misunderstandings or errors. This issue can be mitigated by discarding all other model-based artifacts as soon as a mature implementation exists. In the best case, it eliminates the possibility of outdated or false information. However, relying on the source code as only Domain Model expression requires it to be readable and expressive. Depending on a project's policy towards documentation and similar artifacts, the approach may not be feasible.

Refactoring

Refactoring stands for the technical improvement of software without changing its functionality. The activity can be focused on the amount of code, on its design, the performance and various other non-functional aspects. However, concrete functionalities and their characteristics must remain unmodified. In the context of DDD, refactoring can stand for another concept that is not exclusively related to code. Rather, it is about the functional refinement of a Domain Model and its implementation. While the original practice aims to improve software in a non-functional way, here the goal is to evolve the Domain Model. This is achieved by refining the existing abstractions and adjusting the associated implementations. Both practices are useful, but they should always be separated strictly from each other.



Types of refactoring

Technical refactoring

Improvement of code quality and non-functional aspects

Domain Model refactoring

Refinement of Domain Model and its implementation

Continuous refinement is vital for a Domain Model and its associated source code. The second chapter explained that modeling should be done in an iterative way. This also applies to the implementation. In general, the concepts of this book do not necessitate any type of refactoring activity. However, not following this practice can have notable effects on a development process. One is that the absence of refactoring requires to create a completely correct Domain Model before starting the development. Also, every abstraction needs to be translated into source code without making an error. Whether such a sequential approach seems feasible and useful, is an individual decision to make. Judging from personal experience, iterative modeling and frequent refactoring are essential key concepts.



Emotional attachment to code

Software developers can get emotionally attached to their own personal creations. In the worst case, the idea of refactoring is rejected due to the fear of deleting source code. Refining the Domain Model must always have a higher priority than the perpetual existence of an implementation.

Readability

High readability is an important quality for good code. In general, developers spend more of their time reading and understanding existing source code than writing new one. Consequently, this activity should be as straightforward and efficient as possible. The source code of a software is, among other things, a textual expression of knowledge. For this purpose, it must be easy to comprehend, especially when it is the only Domain-Model-related artifact. Ideally, Domain Experts can capture and discuss functionalities based on the source code. Having a readable implementation makes it also easier to transfer knowledge, for example when introducing new team members. Although the actual complexity of a problem is unalterable, high readability can prevent its software-based solution from additional overhead.

Aspects of readability

- Explicit and intention-revealing identifiers
- Consistent language and style of writing
- Excellent usage of underlying natural language
- Reading flow through logical order of statements
- Optimal length and abstraction levels of blocks

The degree of readability consists of multiple aspects. At the most basic level, the names of identifiers, such as variables and functions, are important. They should be explicit and intention-revealing. Short, cryptic and abbreviated terms must be avoided. Overall, the used language must stay consistent. Multiple styles of writing and different vocabularies cause confusion. An excellent usage of the underlying natural language is mandatory. Another important aspect is the arrangement of source code. Statements should follow a logical order to ensure a continuous reading flow. The lengths of delimiting software constructs, such as functions or blocks, also affect readability. Their word count should not be too high, and they should only contain one level of abstraction. Long and information-packed sections are generally harder to comprehend.

Example: Interest matching service

Consider implementing a service that calculates a matching score for two collections of interests. This functionality can be used as standalone service or it can be integrated into other software, such as social applications. The service must compare the individual items of two given collections and calculate a score that indicates the matches. While the individual interests can contain arbitrary text, it is assumed that each consists of short and unambiguous terms. This reduces the required complexity for an implementation that produces reliable results. In order to illustrate the advantages of a high readability, the domain-specific score logic is only explained at the end.

The first example provides a correct implementation without paying attention to readability ([run code](#)):

Interest matching service: Unreadable version

```
const calc = (a, b) => {
  const la = a.map(x => x.toLowerCase()), lb = b.map(x => x.toLowerCase());
  const m = Math.max(la.length, lb.length);
  const f = la.reduce((s, x) => s + lb.includes(x), 0) * (100 / m);
  let p = 0;
  la.forEach(x => {
    lb.forEach(y => p += (x !== y && (x.includes(y) || y.includes(x))) * (50 / m));
  });
  return f + p;
};

const calcAndLog = (a, b) => console.log(`[${a}] and [${b}]: ${calc(a, b)}`);

calcAndLog(['chess', 'music'], ['jazz music']);
calcAndLog(['hiking', 'biking'], ['biking']);
calcAndLog(['art', 'yoga'], ['yoga', 'pop art']);
```

The operation `calc()` accepts two collections of interests and returns their matching score. Its source code is short and has a compact style. The continuous reading flow is naturally given. Despite these advantages, the approach has various shortcomings. The naming of identifiers is cryptic, which makes it difficult to anticipate their meaning. This includes the name of the function itself. “`calc`” is an unnecessary abbreviation and not intention-revealing. The variable names inside the function are even worse. Also, there are error-prone optimizations that execute calculations with booleans. Exploiting the implicit type conversion is a bad practice. Overall, the function contains multiple levels of abstraction and has a high complexity. However, the biggest disadvantage is the inability to anticipate the algorithm by reading the code.

The next example provides an implementation that also serves as human-readable expression of knowledge ([run code](#)):

Interest matching service: Readable version

```

const calculateScore = (interestsA, interestsB) => {
  const maximumScore = 100;
  interestsA = interestsA.map(interest => interest.toLowerCase());
  interestsB = interestsB.map(interest => interest.toLowerCase());

  const maxNumberOfInterests = Math.max(interestsA.length, interestsB.length);
  const fullMatchPoints = maximumScore / maxNumberOfInterests;
  const partialMatchPoints = fullMatchPoints / 2;

  const fullMatchCount = getFullMatchCount(interestsA, interestsB);
  const partialMatchCount = getPartialMatchCount(interestsA, interestsB);

  return fullMatchCount * fullMatchPoints + partialMatchCount * partialMatchPoints;
};

const getFullMatchCount = (interestsA, interestsB) =>
  interestsA.filter(interest => interestsB.includes(interest)).length;

const getPartialMatchCount = (interestsA, interestsB) => {
  let countOfMatches = 0;
  interestsA.forEach(interestA => {
    interestsB.forEach(interestB => {
      if (interestA === interestB) return;
      if (interestA.includes(interestB) || interestB.includes(interestA))
        countOfMatches++;
    });
  });
  return countOfMatches;
};

const calculateScoreAndLog = (interestsA, interestsB) => {
  const score = calculateScore(interestsA, interestsB);
  console.log(`[${interestsA}] and [${interestsB}]: ${score}`);
};

calculateScoreAndLog(['chess', 'music'], ['jazz music']);
calculateScoreAndLog(['hiking', 'biking'], ['biking']);
calculateScoreAndLog(['art', 'yoga'], ['yoga', 'pop art']);

```

The operation `calculateScore()` provides the same behavior as the first implementation, but has a high degree of readability. The naming of all functions and variables is explicit

and intention-revealing. There is an excellent and consistent use of the underlying natural language. Instead of exploiting technical mechanisms, all mathematical expressions use actual numbers. Through replacing the usage of the operation `reduce()` with `forEach()`, the code becomes simpler. The main function is divided into multiple subroutines to reduce the contained abstraction levels. This results in an arrangement that makes it possible to read the program from top to bottom. Additional blank lines visualize separate logical sections inside a function. Overall, the code provides a correct implementation and also expresses the underlying model in an understandable way.

With the two implementation approaches in mind, consider the following explanation of the Domain Model for the matching score calculation:

- The minimum score is 0, the maximum score is 100
- Two interests can either match fully or partially
- A full match is when two interests are equal
- A partial match is when one interest contains the text of the other one
- For a full match, the score is increased by a number of points
- This number is equal to 100 divided by the length of the larger collection
- For a partial match, the score is increased by 50% of the points for a full match
- A score of 100 is only possible when all interests fully match

Both approaches provide a correct implementation of the Domain Model. Theoretically, the underlying rules can be extracted by reading either of them. The difference is that the second one is easy to understand, while the first code must be deciphered meticulously. The ultimate goal of readability would be achieved when a non-developer can understand the model by simply reading the code.

Behavior and state

The behavior and the state for the implementation of an individual Domain Model concept form a coherent unit. Consequently, their source code should co-exist in the same space. Without behavior, a state definition merely describes the structure of information that is associated with a Domain concept. On the other hand, behavior cannot be expressed without referring to related state. There is always data to access and eventually new state to be produced. The exact implementation of this principle depends on the respectively applied programming paradigm. When using Object-oriented Programming (OOP), behavior and state should be encapsulated within a common structure, such as a class. In case of Functional Programming (FP), this can be done differently, such as by placing both aspects inside the same module.

Example: Word guessing game

Consider developing the Domain Model implementation for a word guessing game. The game's goal is to determine a word by subsequently guessing individual letters. For that purpose, the state must incorporate the word, the guessed letters and the flag whether the word was guessed. The behavior part of the Domain Model consists of three actions. One is to initiate a game with a given word. The second action is to guess an individual letter. And finally, the implementation must expose the information whether a word was guessed. For the example, the game rules are simplified. There is no try count restriction and no enforcement that each letter is only tried once. Furthermore, nothing prevents from guessing on letters after the game is already finished.

The first code shows an implementation that applies the Object-oriented paradigm ([run code](#)):

Word guessing game: Object-oriented implementation

```
class WordGuessingGame {  
  
    #wordToGuess; #guessedLetters = []; #wasWordGuessed = false;  
  
    constructor(wordToGuess) {  
        this.#wordToGuess = wordToGuess;  
    }  
  
    guessLetter(letter) {  
        this.#guessedLetters.push(letter.toLowerCase());  
        this.#wasWordGuessed = this.#determineIfWordWasGuessed();  
    }  
  
    #determineIfWordWasGuessed() {  
        return this.#wordToGuess.split('').every(  
            letter => this.#guessedLetters.includes(letter.toLowerCase()));  
    }  
  
    wasWordGuessed() { return this.#wasWordGuessed; }  
  
}  
  
const wordToGuess = 'zoo';  
const wordGuessingGame = new WordGuessingGame(wordToGuess);  
wordGuessingGame.guessLetter('z');  
wordGuessingGame.guessLetter('o');  
console.log(`guessed "${wordToGuess}"? ${wordGuessingGame.wasWordGuessed()}`);
```

The next example shows an alternative implementation that applies principles of Functional Programming ([run code](#)):

Word guessing game: Functional implementation

```
const WordGuessingGameState = function(
  {wordToGuess, guessedLetters = [], wasWordGuessed = false}) {
  Object.assign(this, {wordToGuess, guessedLetters, wasWordGuessed});
}

const wasWordGuessed = (word, guessedLetters) =>
  word.split(' ').every(letter => guessedLetters.indexOf(letter.toLowerCase()) > -1);

const guessLetter = (state, letter) => {
  const guessedLetters = state.guessedLetters.concat([letter]);
  return new WordGuessingGameState({
    wordToGuess: state.wordToGuess,
    guessedLetters,
    wasWordGuessed: wasWordGuessed(state.wordToGuess, guessedLetters),
  });
};

const wordToGuess = 'zoo';
const initialState = new WordGuessingGameState({wordToGuess});
const stateAfterGuessingZ = guessLetter(initialState, 'z');
const stateAfterGuessingZO = guessLetter(stateAfterGuessingZ, 'o');
console.log(`guessed "${wordToGuess}"? ${stateAfterGuessingZO.wasWordGuessed}`);
```

The class `WordGuessingGame` implements the game using an object-oriented approach. As constructor argument, it expects a word to guess. All state information is encapsulated as private fields. The operation `guessLetter()` modifies the internal instance state. Determining whether a word was guessed is done by executing the operation `wasWordGuessed()`. The exemplary usage shows that the state is not exposed to the consumer. In contrast, the second implementation uses an FP approach. Here, the state is represented as the constructor `WordGuessingGameState`. Creating a new game is achieved by instantiating it. The operation `guessLetter()` accepts a current state and a letter, and yields a new state. For determining whether a word was guessed, it executes the function `wasWordGuessed()`. The usage code shows that the state is handled outside.

While the two implementations differ in their programming paradigms, they both treat behavior and state as a connected unit. In case of OOP, this is achieved with an enclosing

class. In case of FP, the implementation expresses their relation by simply placing the two aspects next to each other.

Dealing with dependencies

The implementation of an individual concept must be primarily focused on the behavior and data specific to its context. This means that the according code should remain free of details from other concepts as far as possible. Strictly excluding foreign aspects has various advantages. For one, the isolation of code promotes correctness and a high expressiveness. Secondly, it tends to prevent the contained complexity from exceeding the one of the underlying problem. Keeping responsibilities at a minimum also leads to a lower vulnerability to errors. The overall readability is potentially increased, making it easier to utilize the code as textual knowledge expression. Also, there are fewer sources of possible distractions due to issues that are unrelated to the main concept.

Dependency Inversion

The **Dependency Inversion** principle provides an approach to alter the dependency direction of software parts through the use of abstractions. Instead of one part being dependent on another, both parts depend on a common abstraction. Only at runtime, the concrete dependency is passed into the consuming functionality from the outside. Even more, the abstractions can be defined in a way so that the dependency direction is fully inverted. Typically, Dependency Inversion is realized with abstract types that are defined through interfaces. However, JavaScript does not provide a native language construct for this concept. Still, the principle can be applied by keeping the signatures of concrete implementations free of internal details. This achieves that the consumers depend on an implicit abstract interface.

Example: Commenting System

Consider implementing a commenting system that can be integrated into other software. The main functionality is to append a comment to a certain collection. Each individual submission consists of an author name and a message. As there is no authentication involved, there is a high chance of receiving inappropriate content. In order to avoid entries with offensive language, submitted input must be checked before accepting it. Messages that contain disallowed terms must be rejected. Whether this language detection belongs to the Domain Model or not depends on the boundaries of the Domain. For the example, it is

considered to be a foreign concept that is consumed by the commenting system. In fact, such a functionality can be a dedicated software on its own.

The first example shows the offensive language detection component, which enables to search content for a list of specified terms:

Commenting System: Offensive language detector

```
class OffensiveLanguageDetector {  
  
    #offensiveTerms;  
  
    constructor(offensiveTerms) {  
        this.#offensiveTerms = offensiveTerms;  
    }  
  
    doesTextContainOffensiveLanguage(text) {  
        const lowercaseText = text.toLowerCase();  
        return this.#offensiveTerms.some(term => lowercaseText.includes(term));  
    }  
}
```

The next code provides an implementation of a comment collection with the offensive language detector as direct dependency ([run code](#)):

Commenting System: Direct dependency

```
class CommentCollection {  
  
    #comments = [];  
    #offensiveLanguageDetector = new OffensiveLanguageDetector(['stupid', 'idiot']);  
  
    submitComment(author, message) {  
        if (this.#offensiveLanguageDetector.doesTextContainOffensiveLanguage(message))  
            throw new Error('offensive language detected, message declined');  
        this.#comments.push({author, message});  
    }  
  
      
const commentCollection = new CommentCollection();  
commentCollection.submitComment('John', 'This is a great article!');  
commentCollection.submitComment('Jane', 'You are an idiot!');  
commentCollection.submitComment('Joe', 'What about KISS (Keep it simple stupid)?');
```

The class `CommentCollection` defines a changeable collection of comments, to which new entries can be appended to. For the input validation, the component autonomously instantiates the class `OffensiveLanguageDetector` with a list of selected terms. The operation `submitComment()` is responsible for validating and conditionally adding a comment to the internal collection. As first step, the given message is passed to the function of the offensive language detection. In case of disallowed terms, an according exception is thrown. If the message is free of offensive language, the comment is added to the collection. The main issue with this implementation is that the class `CommentCollection` directly depends on the foreign component `OffensiveLanguageDetector`. Ideally, it should only depend on an abstract interface without knowing about concrete external functionalities.

The following example shows an implementation of the commenting system with the offensive language detector as abstract dependency ([run code](#)):

Commenting System: Inverted dependency

```
class CommentCollection {  
  
    #comments = [];  
    #offensiveLanguageDetector;  
  
    constructor({offensiveLanguageDetector}) {  
        this.#offensiveLanguageDetector = offensiveLanguageDetector;  
    }  
  
    submitComment(author, message) {  
        if (this.#offensiveLanguageDetector.doesTextContainOffensiveLanguage(message))  
            throw new Error('offensive language detected, message declined');  
        this.#comments.push({author, message});  
    }  
  
}  
  
const terms = ['stupid', 'idiot'];  
const offensiveLanguageDetector = new OffensiveLanguageDetector(terms);  
const commentCollection = new CommentCollection({offensiveLanguageDetector});  
commentCollection.submitComment('John', 'This is a great article!');  
commentCollection.submitComment('Jane', 'You are an idiot!');  
commentCollection.submitComment('Joe', 'What about KISS (Keep it simple stupid)?');
```

The reworked implementation of the class `CommentCollection` only depends on an abstract interface for the offensive language detection. This is done by introducing a constructor

that accepts a concrete component to be passed in from the outside. The code for the main operation `submitComment()` is identical with the first approach. Still, it only depends on an abstract component. The usage code demonstrates this characteristic. The class `OffensiveLanguageDetector` is instantiated on the outside with a selected list of terms. Only when creating an instance of the component `CommentCollection`, the concrete functionality is passed in as constructor argument. With this implementation approach, the domain-specific component remains free of foreign details. Instead, an implicit abstraction is introduced, on which both components depend upon.



Dependency Injection

The act of passing in a concrete dependency from the outside is called **Dependency Injection**. This can be done in different ways. The example uses the so-called Constructor Injection. Another option is Property Injection, where a dependency is directly assigned to a property from the outside. The third possibility is to pass in dependencies as additional arguments to specific operations.

Command-Query-Separation

Command-Query-Separation (CQS) is a programming principle according to which each function must either be a command or a query. Commands execute actions and modify state, but never return a value. Queries perform computations and return data, but must not change state. They are referentially transparent and execute without side effects. In contrast, every command invocation must be expected to modify state. Although this differentiation happens on a low abstraction level, it fosters cleanly separated and predictable code designs. The respective type of each function should be expressed through an intention-revealing name. Commands must be written as such, for example `writeMessage(author, text)` or `guessLetter(letter)`. Queries may use an imperative style like `calculateScore(interestsA, interestsB)`, but should be phrased as questions for boolean results, such as `wasWordGuessed()`.

Recommendation and exceptions

CQS can be considered generally useful for any part of a software. More specifically, most Domain Model implementations are likely to benefit from this principle. In theory, every functionality can be implemented with pure commands and queries. However, there are practical exceptions. Therefore, besides the expected design improvements, other effects

such as additional complexity or error-proneness must be considered. There are even native JavaScript interfaces that combine both function types. For example, the operation `Array.pop()` removes an element and also returns it. Here, the code aims to guarantee that the performed process is atomic. Otherwise, it would be hard to ensure that the same item is returned as is removed. Nevertheless, for many scenarios, the use of CQS helps to improve the code design.

Example: Newsletter subscription

For this example, consider implementing a newsletter component for topic-based subscriptions. The essential behavior the Domain Model defines is the ability to subscribe to a single topic. For multiple topics, multiple subscription actions must be performed. Whenever a subscription is attempted, both the e-mail address and the topic must be checked for their validity. There are three domain-specific rules to enforce in the implementation. First, the e-mail address must be valid. For brevity reasons, it is assumed that this rule is satisfied by verifying the existence of an “@” character. Secondly, the e-mail address must not already be subscribed to the selected topic. Finally, a selected topic needs to be valid value of the list that a newsletter has to offer.



Meaning of the term validation

The term validation can stand for different things and its respective meaning heavily depends on the context. Generally, it can be about user inputs, code contracts, business rules and other aspects. For this example, it is about domain-specific validation of input data.

The first example shows an implementation that does not make use CQS ([run code](#)):

Newsletter subscription: Without CQS

```
class Newsletter {  
  
    #subscribersByTopic = {};  
  
    constructor(topics = []) {  
        topics.forEach(topic => this.#subscribersByTopic[topic] = []);  
    }  
  
    subscribe(emailAddress, topic) {  
        const errors = [];  
  
        if (!emailAddress.includes('@')) {  
            errors.push(`Email address ${emailAddress} is not valid`);  
        }  
  
        if (this.#subscribersByTopic[topic].includes(emailAddress)) {  
            errors.push(`Topic ${topic} is already subscribed`);  
        }  
  
        if (errors.length === 0) {  
            this.#subscribersByTopic[topic].push(emailAddress);  
        }  
        return errors;  
    }  
}
```

```
const subscribers = this.#subscribersByTopic[topic];
if (!emailAddress.includes('@')) errors.push('invalid e-mail');
if (!subscribers) errors.push('invalid topic');
else if (subscribers.includes(emailAddress)) errors.push('duplicate');
if (errors.length === 0) subscribers.push(emailAddress);
return {validationErrors: errors};
}

}

const codingNewsletter = new Newsletter(['html', 'css', 'js']);
const result1 = codingNewsletter.subscribe('john.doe@example.com', 'html');
const result2 = codingNewsletter.subscribe('john.doe@example.com', 'html');
const result3 = codingNewsletter.subscribe('John Doe', 'java');
console.log('subscribe results:\n', [result1, result2, result3]);
```

The class `Newsletter` implements a topic-based newsletter subscription mechanism. As constructor argument, it expects a list of topics. The function `subscribe()` is responsible for adding an individual subscription. First, it validates and verifies given input data. All encountered errors are gathered and returned as validation result. This is the query part of the operation. However, the function name “subscribe” does not clearly express that there is a return value. Also, there is no possibility to validate input data without attempting to subscribe. If the data is valid, the state is modified by adding the given e-mail as subscriber to the selected topic. This is the command part of the operation. While the implementation provides a correct behavior, it violates CQS and has too many responsibilities.

Instead of returning validation results, the next implementation throws an exception when encountering invalid input ([run code](#)):

Newsletter subscription: With exceptions

```
class Newsletter {

    #subscribersByTopic = {};

    constructor(topics = []) {
        topics.forEach(topic => this.#subscribersByTopic[topic] = []);
    }

    subscribe(emailAddress, topic) {
        const subscribers = this.#subscribersByTopic[topic];
        if (!emailAddress.includes('@')) throw new Error('invalid e-mail');
```

```
    if (!subscribers) throw new Error('invalid topic');
    else if (subscribers.includes(emailAddress)) throw new Error('duplicate');
    subscribers.push(emailAddress);
}

const tryCatch = operation => {
  try { operation(); } catch (error) { console.log(error); }
};

const codingNewsletter = new Newsletter(['html', 'css', 'js']);
tryCatch(() => codingNewsletter.subscribe('john.doe@example.com', 'html'));
tryCatch(() => codingNewsletter.subscribe('john.doe@example.com', 'html'));
tryCatch(() => codingNewsletter.subscribe('Jon Doe', 'java'));
```

The reworked class `Newsletter` is similar to the first implementation, but differs in the way validation issues are handled. Its function `subscribe()` has no return value and instead throws exceptions for invalid inputs. This approach seemingly looks like a solution that applies CQS correctly. However, it is in fact worse. The code still has the same responsibilities. Only the mechanism how to yield a validation result is different. Also, the function name does not express that there may be exceptions, which can cause unexpected crashes. Exceptions must be used for exceptional cases, not for expected situations. Another disadvantage is that it only reports the first encountered error. This is a deterioration of the code design. Using exceptions for validation is generally a bad programming practice.



Exceptions versus errors

The terms “exception” and “error” are often used interchangeably in software development. This may be due to the lack of a standardized definition. JavaScript exclusively provides a native type for errors. This book uses the term “exception” in the written text, but refers to “errors” as `Error` instances in the code.

The final implementation shows an approach that correctly applies CQS ([run code](#)):

Newsletter subscription: With CQS

```
class Newsletter {  
  
    #subscribersByTopic = {};  
  
    constructor(topics = []) {  
        topics.forEach(topic => this.#subscribersByTopic[topic] = []);  
    }  
  
    validateSubscription(emailAddress, topic) {  
        const errors = [];  
        const subscribers = this.#subscribersByTopic[topic];  
        if (!emailAddress.includes('@')) errors.push('invalid e-mail');  
        if (!subscribers) errors.push('invalid topic');  
        else if (subscribers.includes(emailAddress)) errors.push('duplicate');  
        return {containsErrors: errors.length > 0, errors};  
    }  
  
    subscribe(emailAddress, topic) {  
        const validationResult = this.validateSubscription(emailAddress, topic);  
        if (validationResult.containsErrors) throw new Error('invalid arguments');  
        this.#subscribersByTopic[topic].push(emailAddress);  
    }  
  
}  
  
const codingNewsletter = new Newsletter(['html', 'css', 'js']);  
  
const emailAddress1 = 'john.doe@example.com', topic1 = 'html';  
const result1 = codingNewsletter.validateSubscription(emailAddress1, topic1);  
if (!result1.containsErrors) codingNewsletter.subscribe(emailAddress1, topic1);  
  
const emailAddress2 = 'john.doe@example.com', topic2 = 'html';  
const result2 = codingNewsletter.validateSubscription(emailAddress2, topic2);  
if (!result2.containsErrors) codingNewsletter.subscribe(emailAddress2, topic2);  
  
const emailAddress3 = 'John Doe', topic3 = 'java';  
const result3 = codingNewsletter.validateSubscription(emailAddress3, topic3);  
if (!result3.containsErrors) codingNewsletter.subscribe(emailAddress3, topic3);  
  
console.log('subscribe results:\n', [result1, result2, result3]);
```

This approach adheres to the CQS principle without decreasing code quality or misusing tech-

nical concepts. The intention-revealing query `validateSubscription()` returns a validation result for a given input without attempting an actual subscription. In contrast, the command `subscribe()` mutates state by adding an e-mail to a selected newsletter topic. However, the function also contains code that again seemingly mixes concerns and misuses exceptions. Prior to the mutation, it executes `validateSubscription()` and throws an exception in case of validation errors. For this approach, it is important to understand the intended component usage. Consumers are responsible for validating their input before subscribing. Therefore, passing invalid input to the command is an unexpected case. Calling the validation and conditionally throwing an exception ensures the integrity of the newsletter state.

The illustrated aspects and principles of this chapter are low-level building blocks that help to achieve a high code quality. The next step is to look at the patterns that enable to express Domain Model concepts as concrete source code.

Chapter 6: Value Objects, Entities and Services

The implementation of a Domain Model is an expression of abstractions as source code, targeted to solve specific problems. If this part does not manage to fulfill its duties appropriately, the whole surrounding project is likely to fail its purpose. Therefore, it should be considered the heart of every software. DDD provides a number of tactical patterns that guide the code design of the Domain layer. These patterns can be applied independently of the concept DDD as a whole. At a minimum, a Domain Model implementation should consist of a combination of Value Objects and Entities. On top of that, stateless functionalities can be designed as Domain Services. Applying these tactical patterns helps to express individual Domain Model concepts adequately.



Type checking and validation

As explained in the Preface, the code for this book makes only limited use of type checking and data validation. In this chapter, both of the mechanisms are used sporadically for protecting invariants and ensuring integrity in Domain Model implementations.

Value Objects

The implementation of a Domain Model component is typically either done as Value Object or as Entity. **Value Objects** quantify or describe an aspect of a Domain without conceptual identity. They are defined by their attributes. [Evans, p. 98] describes them as elements “that we care about only for what they are, not who or which they are”. For example, two date objects are considered the same as long as their day, month and year match. According to [Vernon, p. 219], such objects are “easier to create, test, use, optimize and maintain”. Therefore, they should be used extensively. Value Objects must provide a few key characteristics. The most important ones are explained in the following subsections. This is preceded by the introduction of an overarching example topic.



Value Objects versus data

Value Objects may have functions and carry out domain-specific behavior. They are first class citizens of a Domain Model implementation and must not be confused with plain data structures. The subsection on Immutability provides an example to illustrate such meaningful behavior.

Example: Floor planning software

Consider implementing selected parts of the Domain Model for a floor planning software. Its purpose is to enable virtual room layout plannings. For the example, two selected concepts of the Domain Model are implemented. One is the furniture component, which consists of a type, a width and a length. The type is represented as a simple string. In contrast, both the width and the length are so-called measurements, which is the second relevant Domain Model part. The measurement concept is defined as a combination of magnitude and unit. While the magnitude is a plain number, the unit is an element out of a list of fixed values. In the following subsections, both components are implemented in different ways to illustrate the respective Value Object characteristics.

Conceptual Whole

A Value Object is a collection of coherent attributes and related functionality that form a Conceptual Whole. This means, it is a self-contained structure that captures some descriptive aspect of a Domain. The consequential boundary makes it possible to give an intention-revealing name based on the respective Ubiquitous Language. In most scenarios, attributes that belong to a common conceptual unit should not be placed separately on their own. Doing so fails to capture the idea of the Domain Model. At the same time, the attributes must not be embedded into larger structures that are focused on other concepts. Otherwise, the code establishes incorrect boundaries that do not align with the actual abstractions. Related attributes and their associated actions should be combined into meaningful Value Objects.

The first example shows standalone attributes that together represent a single furniture instance ([run code](#)):

Floor Planning: Standalone attributes

```
const furnitureType = 'desk';
const furnitureWidth = 100, widthUnit = 'cm';
const furnitureLength = 60, lengthUnit = 'cm';

console.log(`dimensions of ${furnitureType} furniture:`);
console.log(`${furnitureWidth}${widthUnit} x ${furnitureLength}${lengthUnit}`);
```

The second example provides a class that expresses the furniture concept as Value Object type ([run code](#)):

Floor Planning: Furniture Value Object

```
class Furniture {

    type; length; lengthUnit; width; widthUnit;

    constructor({type, length, lengthUnit, width, widthUnit}) {
        Object.assign(this, {type, length, lengthUnit, width, widthUnit});
    }

    toString() {
        return `${this.type}, ${this.width}${this.widthUnit} wide` +
            `, ${this.length}${this.lengthUnit} long`;
    }
}

const desk = new Furniture(
    {type: 'desk', length: 60, lengthUnit: 'cm', width: 100, widthUnit: 'cm'});
console.log(desk.toString());
```

The first approach using standalone attributes does not express the domain-specific concepts furniture and measurement in a meaningful way. There is no structural relationship or enclosing boundary across the individual values. Not even their variable naming unambiguously states that there is a conceptual connection between them. In contrast, the second implementation correctly captures the furniture concept as the class `Furniture`. The code defines a structure that encloses related aspects. However, it mixes multiple concepts into a single unit and therefore establishes an incorrect boundary. This is because the furniture component must not be concerned with measurement details. Rather, they must

be encapsulated into a separate Value Object type. Embedding them directly makes it less clear what the intrinsic attributes of furniture instances are.

The next example implements both the furniture and the measurement concept as Value Object type ([run code](#)):

Floor Planning: Measurement Value Object

```
class Measurement {  
  
    magnitude; unit;  
  
    constructor({magnitude, unit}) {  
        Object.assign(this, {magnitude, unit});  
    }  
  
    toString() {  
        return `${this.magnitude}${this.unit}`;  
    }  
  
}  
  
class Furniture {  
  
    type; length; width;  
  
    constructor({type, length, width}) {  
        Object.assign(this, {type, length, width});  
    }  
  
    toString() {  
        return `${this.type}, ${this.width} wide, ${this.length} long`;  
    }  
  
}  
  
const width = new Measurement({magnitude: 100, unit: 'cm'});  
const length = new Measurement({magnitude: 60, unit: 'cm'});  
const desk = new Furniture({type: 'desk', length, width});  
console.log(desk.toString());
```

The class `Measurement` accurately expresses the Domain Model definition of a measurement as a dedicated Value Object type. Instead of maintaining individual attributes, the component

Furniture uses measurement objects for both the width and the height. This allows the class to focus on its own specific attributes. Aspects that are only relevant to details of measurement components do not affect the state of furniture. For example, consider changing a desk's width from 100 centimeters to 120 centimeters. With the previous approach, this requires multiple attributes inside the furniture to change. When using separate self-contained measurement Value Objects, the state of the enclosing furniture remains largely unaffected. The measurement of 100 centimeters is simply replaced by a new one representing 120 centimeters.

Equality by Values

Two Value Objects of the same type are considered equal when all their corresponding attributes have equal values. Consequently, it is also irrelevant whether they are represented by the same reference object. This characteristic is closely tied to the absence of a conceptual identity. Compare this to primitive types such as strings or numbers. The character sequence "test" is always equal to "test", independent of the technical handling of strings. The number 42 is always equal to 42, even if both values originate from different memory addresses. It is a matter of equality and not identity. The same applies to Value Objects. However, other than primitives, they consist of multiple parts. Therefore, all their attributes need to match for Value Objects to count as equal.

The following code example implements Equality by Value for the previously introduced measurement component ([run code](#)):

Measurement: Equality by Values

```
class Measurement {  
  
    magnitude; unit;  
  
    constructor({magnitude, unit}) {  
        Object.assign(this, {magnitude, unit});  
    }  
  
    equals(measurement) {  
        return measurement instanceof Measurement &&  
            this.magnitude === measurement.magnitude && this.unit === measurement.unit;  
    }  
}
```

```
const desk1Height = new Measurement({magnitude: 60, unit: 'cm'});
const desk2Height = new Measurement({magnitude: 60, unit: 'cm'});
console.log(`are the heights equal? ${desk1Height.equals(desk2Height)})`);
```

The updated implementation of the class `Measurement` provides the operation `equals()` to check whether two Value Objects are equal. As argument, it expects another measurement to compare to. First, the operation verifies that the provided argument is an actual measurement. As described in the beginning of the book, type checking in JavaScript is limited in its possibilities. Depending on the respective context, one particular mechanism can be most useful. Here, the example code uses the `instanceof` keyword to verify a correct constructor. In case of success, the attributes for unit and magnitude are tested for value equality. Finally, the operation returns either `true` or `false`. The usage code demonstrates that two different reference objects are considered equal when both their unit and magnitude match.

Immutability

Value Objects must be immutable. This means that their attributes are not allowed to change after an initial assignment. Consequently, all necessary values must be provided at construction time. The main motivation for this is best understood when comparing it to mathematics. A number never changes and always represents one specific numerical value. An addition of two numbers refers to a third one, but does not cause any modification. Algebraic variables can change the values they refer to, but the values themselves remain constant. The same logic applies to Value Objects. This has some useful advantages. For one, an object that is initially valid always remains valid. There is no later verification or validation required. Secondly, working with an immutable object cannot cause side effects.

The next implementation shows an approach for enforcing Immutability on the example measurement Value Object ([run code](#)):

Measurement: Immutability

```
const Measurement = function({magnitude, unit}) {
  Object.freeze(Object.assign(this, {magnitude, unit}));
};

const deskWidth = new Measurement({magnitude: 100, unit: 'cm'});
try {
  deskWidth.magnitude = 200;
} catch (error) {
  console.log(error.message);
}
```

The implementation of the component `Measurement` represents an immutable Value Object type without dedicated behavior. For this particular example, the component is implemented as plain constructor function. Upon instantiation, both the values for magnitude and unit are assigned to the according attributes. Afterwards, the operation `Object.freeze()` is executed with the newly created instance as argument. This closes the measurement for any kind of future modification, including value assignments. In terms of Immutability, the same result can be achieved with the functions `Object.defineProperty()` or `Object.defineProperties()`. Even more, when only parts of an object should be immutable, those operations must be used. The exemplary usage demonstrates that any attempt to modify an attribute of a frozen object results in an exception.

Side-effect free behavior

As mentioned earlier, Value Objects may carry out domain-specific operations. While their Immutability seems to make them a poor candidate for behavior, this is only a matter of proper code design. There are different ways to implement a functionality without mutating state. One approach is to create a new instance and replace an existing one. Generally, this reduces the chances of unexpected side effects. For example, consider the requirement to add together two measurements with equal units. The first approach is to alter one measurement by the magnitude of a second one. Another possibility is to create a new instance as a combination of two existing ones, similar to a numerical addition. In fact, measurements themselves do not change, only the consumers change what they reference.

The following example provides a measurement component with an operation to add multiple instances together ([run code](#)):

Measurement: Side effect free behavior

```
class Measurement {  
  
    magnitude; unit;  
  
    constructor({magnitude, unit}) {  
        Object.assign(this, {magnitude, unit});  
    }  
  
    plus(measurement) {  
        if (this.unit !== measurement.unit) throw new Error('unit mismatch');  
        const combinedMagnitude = this.magnitude + measurement.magnitude;  
        return new Measurement({magnitude: combinedMagnitude, unit: this.unit});  
    }  
  
}  
  
const deskWidth = new Measurement({magnitude: 100, unit: 'cm'});  
const extension = new Measurement({magnitude: 20, unit: 'cm'});  
const newDeskWidth = deskWidth.plus(extension);  
console.log(newDeskWidth);
```

The class `Measurement` defines the instance operation `plus()` to add two measurements together in a side-effect-free manner. This function first verifies that the passed in object has the same unit as the current instance. In case of a mismatch, it throws an according exception. Instead of requiring identical units, the implementation could be extended with a conversion mechanism that aligns measurements with different units. However, for the example, this extended use case is ignored. If both measurements have equal units, the magnitudes are added together. The `Measurement` constructor is invoked with the sum and the shared unit as arguments. Finally, the new instance is returned. The exemplary usage demonstrates that the addition of two measurements creates a new object without mutating existing ones.

The described characteristics show the numerous advantages of Value Objects. As mentioned previously, the pattern should be used extensively. At the same time, when elements are distinguished by other means than their attributes, the Entity pattern is more suitable.

Entities

Entities are unique elements of a Domain that have a conceptual identity. [Vernon, p. 171] suggests to design a concept “as an entity when we care about its individuality”. In this

case, the elements primarily matter for who or which they are rather than what they are. For example, two persons with equal names and equal addresses may not be the same. Regardless of similarities, they are distinguished by a unique identity. Such an identity must be permanent and constant. Entities are also collections of related attributes and domain-specific behavior. The difference is that they have a lifecycle throughout which they can change. Consequently, two Entities with different values can be the same, only in different versions. Likewise, Entities that are equal may not be the same.

Identities

The identity of an Entity distinguishes it from other elements independent of its state or shape. There are three requirements for an identity. For one, it has to exist during the whole Entity's lifecycle. Secondly, it must be immutable. Third, it has to be unique. However, the uniqueness may only be valid within a specific boundary. [Evans, p. 92] explains that identifiers might be valid "only in the context of the system" or "outside a particular software". At a minimum, there may not be duplicated identifiers within a single conceptual boundary. The most commonly chosen context is the enclosing software. This enables to identify Entities unambiguously across subsystems. When multiple software products have to integrate with each other, it is useful to consider globally unique identifiers.

Natural vs. artificial identifiers

There are mainly two possibilities for the composition of identities. The first one is to utilize an existing immutable attribute or a combination of many. For a user account Entity type, this could be the e-mail address. For an online multiplayer game, it may be the concatenation of username and server name. This approach is useful when identities must be human-readable. However, it can lead to problems when the value of an attribute can change. An e-mail address is a good example of something unique that is not necessarily constant. The second approach is to use artificial identifiers. They introduce additional complexity, but serve a dedicated purpose. Good examples are tax IDs and social security numbers. Generally, this book recommends the use of artificial identifiers.

Identifier generation

Generating identifiers for Entities can be done in different ways. One approach is to let the employed persistence technology create them. Many storage systems are capable of generating unique values. However, there are disadvantages with this approach. Generally, it increases the risk of making the Domain Model implementation dependent on a specific

technology. Furthermore, it often implies that the identity of a new Entity is only accessible after saving it. Another possibility is to generate identifiers in the runtime without relying on external processes. Ideally, this is done in a synchronous way, as it is less complex. This approach makes it even possible to provide identities from outside the Application Layer. Doing so helps to support idempotent Entity creation, which is covered in Chapter 10.

Universally Unique Identifier

Universally Unique Identifier (UUID) is a standard for identifiers that are primarily used in software systems. The proposed structure is a 128 bit long number, which is displayed as 32 hexadecimal digits, separated into 5 groups. The standardized structure eliminates the need for custom formats. Generally, it is a powerful mechanism, as it does not demand a centralized component and can be used in distributed systems. There are different versions of UUIDs. The most commonly used one is version 4, because it exclusively depends on random numbers and does not require namespaces. Independent of the version, the structure is always the same. With this standard, a software can stay independent of specific technologies. As long as the generated values are standards-compliant, the underlying implementation is irrelevant.

Example UUID: `4a67e856-441e-431f-9c11-75863d51cbe`

The likelihood of a collision for UUIDs primarily depends on the quality of the random number generation (RNG). `Math.random()` is one commonly used option for creating random values. However, the mechanism only produces pseudo-random numbers that have a relatively high chance of duplicates. The Web Cryptography API standard defines the operation `crypto.getRandomValues()` to produce cryptographically secure random numbers. This standard is implemented in recent versions of most browsers as well as in Node.js. While an identifier collision is theoretically still possible with this mechanism, the chances are very low. Even more, since the identity uniqueness is often bound to a certain context, the risk of duplicates is negligible.

The following example provides an operation that creates UUID-compliant identifiers ([run code](#)):

ID generation: Generate id function

```
const uuidV4Template = '????????-????-4???-1??-????????????';  
  
const generateId = () =>  
  uuidV4Template.replace(/\?/g, () => Math.round(Math.random() * 15).toString(16));
```

The operation `generateId()` creates an artificial identifier that is compliant to the UUID standard version 4. As random number generation mechanism, `Math.random()` is used. For a correct format, it defines a string template. This template contains question marks as placeholders for the actual random values. In addition to that, it incorporates two fixed numbers. The digit 4 is placed as the M field, which defines the UUID version. The value 1 is used for the N field, which is the UUID variant. Strictly speaking, only the first 3 bits of the hexadecimal digit are reserved for the variant. However, for simplicity reasons, the implementation assumes that the fourth bit is always 0. The function `generateId()` is used throughout all following code examples in the book.

Example: Address book

Consider implementing a part of the Domain Model for an address book software. The purpose is to maintain a list of personal contacts. Every contact consists of several details, of which each can change over time. The example focuses on the contacts component without dealing with the surrounding address book concept. The required attributes for a contact are first name, last name, street, house number, zip code and country. All the attributes are represented as simple strings. Theoretically, the combination of a name and a postal address has a high chance of unambiguously identifying a person. Still, the contact component must be designed as Entity type. The main reason is that its attributes can all change over time, while the conceptual identity must remain constant.

The first example shows a basic implementation and usage of a contact Entity component ([run code](#)):

Address book: Contact with flat attributes

```
class Contact {  
  
    id; firstName; lastName; street; houseNumber; zipCode; country;  
  
    constructor({id, firstName, lastName, street, houseNumber, zipCode, country}) {  
        Object.defineProperties(this, {  
            id: {value: id, writable: false},  
            firstName: {value: firstName},  
            lastName: {value: lastName},  
            street: {value: street},  
            houseNumber: {value: houseNumber},  
            zipCode: {value: zipCode},  
            country: {value: country},  
        });  
    }  
  
}  
  
const contact1 = new Contact({id: generateId(), firstName: 'John', lastName: 'Doe',  
    street: 'First Street', houseNumber: '4', zipCode: 19061, country: 'Germany'});  
const contact2 = new Contact({id: generateId(), firstName: 'John', lastName: 'Doe',  
    street: 'Second Street', houseNumber: '1', zipCode: 19061, country: 'Germany'});  
  
contact2.houseNumber = '4';  
contact2.street = 'First Street';  
  
console.log(contact1, contact2);
```

The class `Contact` expresses the domain-specific concept of a contact for an address book. All its attributes are defined as public class fields. The constructor accepts an argument for each of them and assigns the values using the operation `Object.defineProperties()`. For the identifier, the according field is configured to be immutable. This creates contact Entities with a constant identifier and mutable attributes. The usage code creates two `Contact` instances that only differ in parts of their address. Afterwards, both the street and the house number of one instance is changed to be the same as the other. The usage code demonstrates two aspects. For one, an Entity retains its identity, regardless of state. Secondly, two Entities with identical attributes are not necessarily the same.

Value Containers

Entities should have as few direct attributes as possible and act as so-called value containers. Instead of enclosing many individual and possibly unrelated attributes directly, they should incorporate Value Objects whenever possible. The main responsibility of Entities is to manage their identity and their lifecycle. As a consequence, they should not be concerned with additional details of subordinate concepts. This design goal aligns well with the idea of encapsulating self-contained descriptive concepts as Value Objects whenever possible. As an example, consider a cooking recipe Entity. The element should not have direct attributes for components and amounts. Rather, it should act as a container that encloses ingredients, which themselves are composite Value Objects.

Example: Address book improvements

Consider improving the previously illustrated implementation for the contact Entity type of the address book software. While the first approach expresses the Domain Model fully and provides the expected behavior, the code design is not ideal. The class Contact has numerous individual and partially unrelated attributes, and therefore too many responsibilities. This design issue can be solved by extracting meaningful concepts into separate components. Both a composite for a full name and a postal address are good candidates for individual Value Object types. This is because they are primarily defined by their attributes, naturally equal by values and can be treated as immutable. Also, they represent generic concepts that are widely understood, even without the context of the example software Domain.

The next code shows a reworked implementation of the contact Entity component with additional Value Object types ([run code](#)):

Address book: Contact with Value Objects

```
const Name = function({firstName, lastName}) {
  Object.freeze(Object.assign(this, {firstName, lastName}));
};

const Address = function({street, houseNumber, zipCode, country}) {
  Object.freeze(Object.assign(this, {street, houseNumber, zipCode, country}));
};

class Contact {
  id; name; address;

  constructor({id, name, address}) {
```

```
Object.defineProperties(this, {
  id: {value: id, writable: false},
  name: {value: name},
  address: {value: address},
});
}

const contact1 = new Contact({id: generateId(),
  name: new Name({firstName: 'John', lastName: 'Doe'}),
  address: new Address({street: 'First Street',
    houseNumber: '4', zipCode: 19061, country: 'Germany'})});
const contact2 = new Contact({id: generateId(),
  name: new Name({firstName: 'John', lastName: 'Doe'}),
  address: new Address({street: 'Second Street',
    houseNumber: '1', zipCode: 19061, country: 'Germany'})});

contact1.address = new Address(
  {...contact1.address, street: 'First Street', houseNumber: '4'});

console.log(contact1, contact2);
```

The reworked class `Contact` is reduced to three individual attributes. As most important part, it encloses its identity. On top of that, there is one attribute for a name and another one for an address. The constructor functions `Name` and `Address` represent the respective Value Object types. Both of them accept individual attributes for all the information they enclose. Through the execution of the function `Object.freeze()`, the created instances are made immutable. The Entity type is reduced to the bare required minimum of direct attributes. As a side effect, the exemplary usage code illustrates another interesting aspect of the new implementation. Instead of changing individual address attributes, a complete replacement makes the action more similar to a real world use case.



Refactoring the Domain Model

The example also demonstrates how the adaptation of an implementation can imply a change to the Domain Model. By expressing the concepts name and address as dedicated Value Object types, they become explicit parts of the abstractions.

Relationships

Entity-to-Entity relationships can be designed in three ways. The first one is to enclose associated elements completely, similar to when an Entity contains a Value Object. This makes mostly sense when the contained parts conceptually do not exist on their own. The second possibility is to only reference identifiers of foreign components. This approach is useful for relationships to other standalone Entities. Containing such elements risks challenges in combination with persistence. In contrast, referencing immutable identifiers typically avoids those issues. The third approach is to define specialized Value Object types that enclose an identity and a subset of additional Entity attributes. This enables to make required information locally available without enclosing a complete element. However, the redundant data can introduce the need for synchronization.

Bidirectional references

There are situations where two Entities both need to be aware of each other. In many cases, this happens as part of a parent-child relationship. Generally speaking, bidirectional Entity references should be treated with caution. The reason is that they are more complex to deal with than unidirectional associations. Still, this does not mean that they should be avoided at all costs. When specific mutual connections are a part of a Domain Model, they should also appear in the implementation. However, a bidirectional relationship should always be expressed with identifier references. Otherwise, the implementation can face issues such as circular dependencies, especially with regard to the persistence of Entities.

Example: Book favorites

As example, consider implementing the Domain Model for a favorite book list software. Overall, the use cases consist of creating favorite lists, adding individual book entries and retrieving the currently favored ones. The book component incorporates an “International Standard Book Number” (ISBN), a title and a description. ISBNs are a form of artificial identifiers and can be therefore be used as conceptual identities. Their presence makes the book concept automatically an Entity type. The list component is defined as a mutable collection of books. Since lists may change over time, and they are not considered equal by values, they are also Entities. The remaining code design question is how to model the relationship between books and lists.

The following code provides the implementation of the book Entity type:

Book favorites: Book Entity

```
class Book {  
  
    isbn; title; description;  
  
    constructor({isbn, title, description = ''}) {  
        Object.defineProperty(this, 'isbn', {value: isbn, writable: false});  
        Object.assign(this, {title, description});  
    }  
  
}
```

The first approach for favorite lists places the book Entities directly inside the lists ([run code](#)):

Book favorites: Entities completely contained

```
class FavoriteBookList {  
  
    id; #books = [];  
  
    constructor({id}) {  
        Object.defineProperty(this, 'id', {value: id, enumerable: true});  
    }  
  
    addBook(book) {  
        this.#books.push(book);  
    }  
  
    getBooks() { return this.#books.slice(); }  
  
}  
  
const book1 = new Book(  
    {isbn: '123-4567890123', title: 'CQRS', description: '...'});  
const bookFavorites = new FavoriteBookList({id: generateId()});  
bookFavorites.addBook(book1);  
console.log('books:', bookFavorites.getBooks());
```

The class `Book` expresses the Domain Model concept of a book as an Entity type. While its ISBN attribute is defined as immutable, both the title and the description are allowed to change. The class `FavoriteBookList` represents the list component. Adding a book entry is

done via the operation `addBook()`. Accessing the currently contained ones is achieved with the function `getBooks()`. One benefit of enclosing complete book Entities is the ability to render them in a human-readable format. However, this approach faces challenges regarding persistence. Conceptually, books exist on their own and one book can be referenced from multiple lists. Containing the Entities completely would result in data redundancy. Furthermore, each change to a book would need to be distributed across all affected lists.

The second approach uses identifier references to express the relationship between lists and books ([run code](#)):

Book favorites: Using identifiers only

```
class FavoriteBookList {  
  
    id; #isbns = [];  
  
    constructor({id}) {  
        Object.defineProperty(this, 'id', {value: id, enumerable: true});  
    }  
  
    addBook(isbn) {  
        this.#isbns.push(isbn);  
    }  
  
    getBooks() { return this.#isbns.slice(); }  
  
}  
  
const book1 = new Book({isbn: '123-4567890123', title: 'CQRS'});  
const bookFavorites = new FavoriteBookList({id: generateId()});  
bookFavorites.addBook(book1.isbn);  
console.log('book identifiers:', bookFavorites.getBooks());
```

The updated component `FavoriteBookList` works only with identifiers and does not enclose complete Entities. This avoids persistence-related challenges with regard to consistency and redundancy. Lists can easily be saved and loaded without special care or custom mechanisms. The referenced identifiers are immutable and therefore always consistent. Also, the interface of the function `addBook()` is simpler because it expects an identifier instead of a full Entity. One downside of this implementation is the lack of required additional information from referenced books. The list component is unable to produce a human-readable output of the contained favorites. This makes the consumer code responsible for

orchestrating both components and assembling the required information. Depending on the respective architecture, this might not be a feasible approach.

The final implementation designs favorite books as dedicated purpose-specific Value Objects ([run code](#)):

Book favorites: Separate Value Object

```
const FavoredBook = function({isbn, title}) {
  Object.freeze(Object.assign(this, {isbn, title}));
};

class FavoriteBookList {

  id; #favoredBooks = [];

  constructor({id}) {
    Object.defineProperty(this, 'id', {value: id, enumerable: true});
  }

  addBook(favoredBook) {
    this.#favoredBooks.push(favoredBook);
  }

  getBooks() { return this.#favoredBooks.slice(); }

}

const book1 = new Book(
  {isbn: '123-4567890123', title: 'CQRS', description: '...'});
const bookFavorites = new FavoriteBookList({id: generateId()});
bookFavorites.addBook(new FavoredBook({isbn: book1.isbn, title: book1.title}));
console.log('favored books:', bookFavorites.getBooks());
```

The constructor function `FavoredBook` represents the Domain Model concept of a favored book as part of a list. Other than the complete `Book` Entity type, it only defines attributes that are relevant for its use case. This frees from the need of enclosing complete Entities in the list component. At the same time, it provides enough data to create a meaningful human-readable output. However, the locally saved information is redundant and becomes outdated as soon as the corresponding Entity changes. Effectively, this approach faces similar challenges in terms of persistence and consistency as the first implementation. Though, there are ways to mitigate this issue. Many of the solution approaches are based on Domain Events, which are covered in the next chapter.

Overall, the example of favorite book lists illustrates and compares the different approaches for modeling Entity-to-Entity relationships. As a general rule of thumb, it is beneficial to use identifier references, unless additional information from foreign Entities is required locally. Chapter 8 and 9 follow up on the considerations from this section and put them into the context of transactions and persistence.

Domain Services

Domain Services expose functionalities that are not tied to a specific thing of a Domain. Most typically, they are used for calculations or other computations that do not carry domain-specific state. Sometimes, such aspects are mistakenly implemented as Value Objects or as Entities. While this does not automatically create an issue, it typically introduces unnecessary complexity to the code. Another use case for Domain Services are functionalities that have dependencies to components outside the Domain Layer. In such cases, all details must be encapsulated inside the implementation and an abstract interface is exposed to the consuming components. The implementation of a Domain Service does not require specific technical constructs. In JavaScript, it can be represented as class, as object or as simple standalone function.



Stateful Services

The absence of state as a requirement for a Domain Service is only focused on domain-related information. Service structures may possess state for technical reasons, such as keeping references to other functionalities they depend on.

Example: Scientific paper submission

Consider implementing an online platform for scientific papers. For the submission of a new paper, an according entry must be created. The entry component consists of a title, an author, an abstract and the actual paper. After the initial submission, only the abstract may be updated. The Domain Model implementation must ensure that an abstract always has a maximum length of 200 words. For that purpose, it must incorporate a mechanism to determine the word count. One possibility is to include this functionality directly in the paper entry component. However, this adds unnecessary responsibilities. The more useful approach is to implement a stateless service for counting the words in a given text. If required, the service can even make use of an external dependency.



More service examples

In [Chapter 5, section “Dealing with Dependencies”](#), there is a more detailed example of a Domain Service. Furthermore, the Sample Application section of this chapter also illustrates a service implementation.

Invariants

In Computer Science, an **Invariant** is a condition that must be true for a certain time span. With regard to Domain Models, it is a constraint that must be satisfied during the complete lifecycle of affected components. [\[Vernon, p. 205\]](#) describes it as “state that must stay transactionally consistent”. The term “transactionally” implies that such a constraint must always be satisfied without delay. As example, consider a minimum password length for user accounts. Values that violate this rule are rejected upfront. Invariants are often explained in the context of Entities, eventually because of their mutable state. Nevertheless, they are equally important for Value Objects, only that their constraints must be verified upon construction. Generally speaking, the concept is about protecting components from entering an invalid state.



Complexity of invariants

Invariants can span across component boundaries and are not restricted to single units. Rather, they can incorporate complex rules and involve a graph of elements. The only requirement for something to count as an invariant is that its rules must be satisfied at all times.

Invariants vs. validation

As mentioned in the example of [Chapter 5, section “Command-Query-Separation”](#), the term “validation” can stand for different things. The meaning heavily depends on the respective context. It can be about verifying user inputs, satisfying code contracts, enforcing business rules and other aspects. In many cases, certain steps of an invariant protection can be seen as validation. Any given input that can cause a state change must be checked for its correctness first. This means that invariants eventually make use of certain validation mechanisms. Still, it would be wrong to say that the two concepts are the same. The difference is that a part being validated may already be in an invalid state. In contrast, an invariant prevents entering such a state in the first place.

Example: Workday planner

Consider implementing the Domain Model of a workday planner software, whose goal is to plan activities for individual workdays. The required behavior is to create workday plans, plan activities and to calculate the remaining time. Overall, the Domain Model consists of two components. One part is the workday plan, which encloses a number of available hours and a list of activities. The second part is the activity itself, which consists of a description and a number of required hours. One invariant is that an activity's description must not exceed 50 characters. For a workday plan, activities can only be added as long as enough hours are left. Consequently, the second invariant is that the remaining hours of a workday plan must not drop below zero.

The following code provides an example implementation of the described Domain Model ([run code](#)):

Workday planner: Domain Model implementation

```
const Activity = function({description, requiredHours}) {
  if (description.length > 50) throw new Error('too long description');
  Object.freeze(Object.assign(this, {description, requiredHours}));
};

class WorkdayPlan {
  id; availableHours; #activities = [];

  constructor({id, availableHours}) {
    Object.defineProperties(this, {
      id: {value: id, writable: false},
      availableHours: {value: availableHours, writable: false},
    });
  }

  planActivity(activity) {
    if (activity.requiredHours > this.getRemainingHours())
      throw new Error(`not enough time remaining for ${activity.description}`);
    this.#activities.push(activity);
  }

  getRemainingHours() {
    const hoursRequiredForTasks = this.#activities
      .map(activity => activity.requiredHours)
      .reduce((hours, taskHours) => hours + taskHours, 0);
  }
}
```

```
    return this.availableHours - hoursRequiredForTasks;
}

getPlannedActivities() { return this.#activities.slice(); }

}

const workdayPlan = new WorkdayPlan({id: generateId(), availableHours: 8});
workdayPlan.planActivity(new Activity({description: 'coding', requiredHours: 4}));
workdayPlan.planActivity(new Activity({description: 'reading', requiredHours: 2}));
console.log('planned activities:', workdayPlan.getPlannedActivities());
console.log('remaining hours:', workdayPlan.getRemainingHours());
workdayPlan.planActivity(new Activity({description: 'taxes', requiredHours: 3}));
```

The constructor function `Activity` expresses the activity concept as a Value Object type. Upon invocation, the operation checks whether the given description exceeds 50 characters and conditionally throws an exception. The class `WorkdayPlan` implements the workday plan concept as an Entity type. Planning a new activity is done via the operation `planActivity()`. This function enforces that the remaining hours are equal or higher than the required hours of the new activity. The check is performed via the operation `getRemainingHours()`, which calculates the difference between the planned hours and the available ones. Note that this function is public and can also be used for upfront validation. The example demonstrates that invariants can either be focused on a single object or on a collection of elements.

Sample Application: Domain Model implementation

This section illustrates the implementations of the Domain Models for the Sample Application. As conceptual foundation, Chapter 1 defines the involved knowledge areas. Based on this, Chapter 2 creates a system of useful abstractions. The structured knowledge is subdivided into conceptual boundaries in Chapter 3. Chapter 4 defines a fitting high-level Software Architecture. The code qualities from Chapter 5 explain how to achieve readability, expressiveness and a clean separation. Finally, the tactical patterns of this chapter make it possible to transform the Domain Models into actual source code. Note that while the following implementations fulfill the requirements of the underlying abstractions, they are not a final solution. Rather, they are to be seen as first approach, which is further refined throughout the book.

Shared functionalities

The initial implementation of the Domain Models is accompanied by three shared functionalities. One is the identifier generation function `generateId()` from the Entity subsection of this chapter. The operation is used for creating UUID-compliant identifiers. The second shared part is the function `verify()` for the verification of a given constraint. This utility helps to avoid code repetition when performing invariant protection. Consequently, it can be seen as syntactic sugar. The third functionality is a mechanism to hash passwords. This is required by the user context for security reasons. The verification helper belongs in the shared Domain directory. In contrast, the other two parts are considered infrastructural functionalities.

The first code example provides the function for a constraint verification ([run code usage](#)):

Verification: Verify function

```
const verify = (constraintName, condition) => {
  if (!condition) throw new Error(`constraint violated: ${constraintName}`);
};
```

The second code example defines an operation to create an MD5 hash for a given value ([run code usage](#)):

Crypto: Create MD5 hash

```
const createMd5Hash = input => crypto.createHash('md5').update(input).digest('hex');
```



Why hash the password?

Passwords must never be stored in plain text. Instead, they should be obfuscated with a hashing mechanism. The validation of a password is done by also hashing the input before comparison. However, note that MD5 is not recommendable because the results can be converted back with a reverse lookup database.

User context

The user context is a good starting point for the Domain Model implementation. There are multiple reasons for this. For one, the associated Generic Subdomain is more commonly known than the other knowledge areas. Therefore, it is easier to directly focus on transforming the abstractions into source code. Another reason to begin with this context is that it does

not have any outgoing dependencies to other areas. This frees the development activity from considerations in terms of relationship and integration. The disadvantage of the approach is that the Core Domain does not receive the highest priority. For the Sample Application, the task board context should ideally be implemented first. Nevertheless, the reversed order makes most sense for applying and illustrating the previously explained patterns.

The Domain Model for the user context defines the two concepts role and user. The role exclusively encloses a name attribute, which is constrained to a set of valid strings. This structure is primarily defined by its attributes and therefore qualifies as Value Object. In contrast, the user concept represents a mutable component that must be identifiable independent of its state. Therefore, it must be implemented as Entity. Besides these two explicit concepts, there are also additional implicit aspects. The e-mail address also qualifies as dedicated Value Object that encloses a string attribute together with an invariant for a valid format. Furthermore, the requirement for unique e-mail addresses necessitates an overarching component. This part must maintain all used e-mail addresses and provide an availability check.



What about unique usernames?

The Domain Model does not define any rules with regard to the uniqueness of usernames. While this seems like a common requirement, assumptions concerning potential rules and invariants must be avoided. Without explicit definition or implicit requirement, there is no need to ensure that a username is only used once.

The following code examples show the implementations of the Value Object types for role ([run code usage](#)) and e-mail address ([run code usage](#)):

User context: Role Value Object

```
const validRoles = ['user', 'admin'];

class Role {

    name;

    constructor(name) {
        verify('role name', validRoles.includes(name));
        Object.freeze(Object.assign(this, {name}));
    }

    equals(role) { return this.name === role.name; }
```

```
}
```

User context: E-Mail address Value Object

```
const emailRegex = /^[^@]+@[^.]+\.[^.]+$/;

class EmailAddress {

    value;

    constructor(value) {
        verify('valid e-mail', emailRegex.test(value));
        Object.freeze(Object.assign(this, {value}));
    }

    equals(emailAddress) { return this.value === emailAddress.value; }

}
```

The next implementation provides a component for maintaining used e-mail addresses ([run code usage](#)):

User context: E-Mail registry

```
const emailAddressesByUser = new Map();

const emailRegistry = {
    setUserEmailAddress(userId, emailAddress) {
        if (!this.isEmailAvailable(emailAddress)) throw new Error('e-mail in use');
        emailAddressesByUser.set(userId, emailAddress);
    },
    isEmailAvailable(emailAddress) {
        const usedEmailAddresses = Array.from(emailAddressesByUser.values());
        return !usedEmailAddresses.some(
            usedEmailAddress => usedEmailAddress.equals(emailAddress));
    },
};
```

The last building block of the user context is the user Entity implementation:

User context: User Entity

```
class User {  
  
    id; #username; #emailAddress; #password; #role; #emailRegistry;  
  
    constructor({id, username, emailAddress, password, role, emailRegistry}) {  
        verify('valid id', id != null);  
        this.#emailRegistry = emailRegistry;  
        Object.defineProperty(this, 'id', {value: id, writable: false});  
        Object.assign(this, {username, emailAddress, password, role});  
    }  
  
    get username() { return this.#username; }  
  
    set username(username) {  
        verify('valid username', typeof username == 'string' && username != null);  
        this.#username = username;  
    }  
  
    get emailAddress() { return this.#emailAddress; }  
  
    set emailAddress(emailAddress) {  
        verify('unused e-mail', this.#emailRegistry.isEmailAvailable(emailAddress));  
        verify('valid e-mail', emailAddress.constructor === EmailAddress);  
        this.#emailAddress = emailAddress;  
        this.#emailRegistry.setUserEmailAddress(this.id, this.#emailAddress);  
    }  
  
    get password() { return this.#password; }  
  
    set password(password) {  
        verify('valid password', typeof password == 'string' && !password);  
        this.#password = password;  
    }  
  
    isPasswordMatching(password) { return this.password === password; }  
  
    get role() { return this.#role; }  
  
    set role(role) {  
        verify('valid role', role.constructor === Role);  
        this.#role = role;  
    }  
}
```

}

The class `Role` represents the user role concept as Value Object type. It exposes an immutable name attribute, which is constrained by the list of valid role names. The class `EmailAddress` captures the concept of an e-mail address. Verifying a correct value is done with a simplified regular expression. Maintaining used e-mail addresses is the responsibility of the service `emailRegistry`. The availability of a value can be checked via the operation `isEmailAvailable()`. The class `User` represents the user Entity and mainly acts as a value container. Its internal attributes are guarded by setter functions to ensure correct types and protect invariants. The artificial identifier makes instances distinguishable independent of state. For ensuring unique e-mail addresses, the e-mail registry is injected upon construction and used subsequently.

The last code example of this subsection illustrates a usage of the previously implemented parts ([run code](#)):

User context: Usage example

```
const userId = generateId(), role = new Role('user');
const emailAddress1 = new EmailAddress('john.doe@example.com');
const emailAddress2 = new EmailAddress('john.doe.81@example.com');

const user = new User({id: userId, username: 'johndoe', emailAddress: emailAddress1,
    password: createMd5Hash('mypassword'), role, emailRegistry});

user.username = 'johndoe81';
user.emailAddress = emailAddress2;
console.log(user);
console.log('is password correct?',
    user.isPasswordMatching(createMd5Hash('mypassword')));

console.log('attempt to create user with identical e-mail address');

new User({id: generateId(), username: 'johndoe2', emailAddress: emailAddress2,
    password: createMd5Hash('mypassword2'), role, emailRegistry});
```

Project context

For illustration purposes, it makes sense to continue with the project context. This conceptual boundary encloses knowledge abstractions that belong to the Project Management Core

Domain. Therefore, it is concerned with aspects that are more specialized than the user context. Furthermore, this area is involved in all four context-overarching relationships. Three of them are outbound connections. One is the reference from projects to task boards. The second relationship is the assignment of users as project owners. Third, each team member references a user identity. In contrast, the fourth relationship is an inbound connection, which defines that task assignees are team members. Amongst other things, these overarching relationships express the need for the involved parts to have conceptual identities.

The Domain Model of the project context contains four parts. The role concept exclusively encloses a non-empty string. Although there is no logical invariant, this part is best implemented as Value Object. The team member concept represents a component that must be referenced from the outside. Consequently, it must be an Entity type. Regardless of the associated user identity, it requires a dedicated identifier. This enables the task board context to stay independent of user details. Also, it makes more sense as users can represent different members in different teams. The team concept encloses a mutable collection of members and also qualifies as Entity. Although a project contains relatively constant attributes, it must be identifiable independent of state. Therefore, it is also implemented as Entity.



Why the separate Team Entity?

While team members could be direct parts of a project, a separate team Entity ensures a clean separation of concerns. Projects are responsible for their name, team and task board. Teams are concerned with managing their members. These two aspects should be separate from each other.

The first implementation provides the project-specific role Value Object ([run code usage](#)):

Project context: Role Value Object

```
class Role {  
    name;  
  
    constructor(name) {  
        verify('valid role', typeof name == 'string' || !!name);  
        Object.freeze(Object.assign(this, {name}));  
    }  
  
    equals(role) { return this.name === role.name; }  
}
```

The next example implements the Entity type for a team member ([run code usage](#)):

Project context: Team member Entity

```
class TeamMember {  
  
    id; userId; #role;  
  
    constructor({id, userId, role}) {  
        verify('valid id', id != null);  
        verify('valid user id', userId != null);  
        Object.defineProperty(this, 'id', {value: id, writable: false});  
        Object.defineProperty(this, 'userId', {value: userId, writable: false});  
        this.role = role;  
    }  
  
    get role() { return this.#role; }  
  
    set role(role) {  
        verify('valid role', role.constructor === Role);  
        this.#role = role;  
    }  
}
```

This is followed by the implementation of the team Entity ([run code usage](#)):

Project context: Team Entity

```
class Team {  
  
    id; #teamMemberIds = [];  
  
    constructor({id}) {  
        verify('valid id', id != null);  
        Object.defineProperty(this, 'id', {value: id, writable: false});  
    }  
  
    addMember(teamMemberId) {  
        verify('team member is new', !this.#teamMemberIds.includes(teamMemberId));  
        this.#teamMemberIds.push(teamMemberId);  
    }  
  
    removeMember(teamMemberIdToRemove) {
```

```

    const indexToRemove = this.#teamMemberIds.indexOf(teamMemberIdToRemove);
    if (indexToRemove === -1) return;
    this.#teamMemberIds.splice(indexToRemove, 1);
}

get teamMemberIds() { return this.#teamMemberIds.slice(); }

}

```

The last code provides the implementation of the project Entity type:

Project context: Project Entity

```

class Project {

    id; #name; ownerId; teamId; taskBoardId;

    constructor({id, name, ownerId, teamId, taskBoardId}) {
        verify(id && name && ownerId && teamId && taskBoardId, 'valid project data');
        Object.defineProperties(this, {
            id: {value: id, writable: false},
            ownerId: {value: ownerId, writable: false},
            teamId: {value: teamId, writable: false},
            taskBoardId: {value: taskBoardId, writable: false},
        });
        this.name = name;
    }

    get name() { return this.#name; }

    set name(name) {
        verify('valid name', typeof name === 'string' && name.length > 0);
        this.#name = name;
    }
}

```

The class `Role` expresses the project-related specialization of a team member as Value Object. Unlike the more specialized invariant of a user role, this structure merely verifies the technical constraint of a non-empty string. The class `TeamMember` expresses the concept of an individual team member. Besides an identifier and a mutable role, its constructor defines the immutable attribute `userId` for referencing a user Entity. The component `Team` encloses a team member collection, consisting of identifier references. There is no need to contain complete Entities. Finally, the `Project` class represents the project concept. Its constructor

defines an identity, a name and identifier references for an owner, a team and a task board. All its attributes except the name are defined as immutable.

The last code shows a basic usage of the project context implementation ([run code](#)):

Project context: Usage example

```
const projectId = generateId(), teamId = generateId(),
  teamMemberId = generateId(), userId = generateId(), taskBoardId = generateId();

const team = new Team({id: teamId});
const project = new Project({id: projectId,
  ownerId: userId, name: 'my project', teamId, taskBoardId});

const teamMember = new TeamMember(
  {id: teamMemberId, userId, role: new Role('developer')});
team.addMember(teamMember.id);
[project, team, teamMember].forEach(item => console.log(item));
console.log(team.teamMemberIds);
```

Task board context

The final Domain Model implementation is for the task board context. This part encloses the key aspects of the Core Domain and has therefore the highest importance for the Sample Application. Most of the expertise and effort must be put into this area. The context is affected by two of the four overarching relationships. One is the inbound connection of projects to task boards. The other one is the optional outbound reference of tasks to project members as assignees. Through the use of separate team member identifiers, it is possible for tasks to be unaware of the user context. This code design ensures a low count of dependencies between Bounded Contexts. Also, changes to the user part only affect the referencing team member component.

The Domain Model of the task board part defines two individual concepts. Apart from these, there are no other aspects to express explicitly. The task part represents a component that encloses multiple mutable attributes and requires a unique identity. These characteristics qualify it as Entity. The task board concept encloses a varying collection of tasks. Due to its changeable state and the necessity to be referenced from outside, it must also be implemented as Entity. One open question is whether a board is responsible for the status of a task or the task itself. The Domain Model defines that a task must be assigned when it is in progress. This invariant can be best protected when the status is a direct attribute of each task.

The first code examples show the implementations for the Entities task ([run code usage](#)) and task board:

Task board context: Task Entity

```
const validStatus = ['todo', 'in progress', 'done'];

class Task {

    id; #title; #description; #status; #assigneeId;

    constructor({id, title, description = '', status = 'todo', assigneeId}) {
        verify('valid id', id != null);
        Object.defineProperty(this, 'id', {value: id, writable: false});
        Object.assign(this, {title, description, status, assigneeId});
    }

    get description() { return this.#description; }

    set description(description) {
        verify('valid description', typeof description == 'string');
        this.#description = description;
    }

    get title() { return this.#title; }

    set title(title) {
        verify('valid title', typeof title == 'string' && !!title);
        this.#title = title;
    }

    get status() { return this.#status; }

    set status(status) {
        verify('valid status', validStatus.includes(status));
        verify('active task assignee', status !== 'in progress' || !!this.assigneeId);
        this.#status = status;
    }

    get assigneeId() { return this.#assigneeId; }

    set assigneeId(assigneeId) {
        verify('active task assignee', this.status !== 'in progress' || assigneeId);
        this.#assigneeId = assigneeId;
    }
}
```

```
 }  
}
```

Task board context: Task board Entity

```
class TaskBoard {  
  
    id; #tasks = [];  
  
    constructor({id}) {  
        verify('valid id', id != null);  
        Object.defineProperty(this, 'id', {value: id, writable: false});  
    }  
  
    addTask(task) {  
        verify('valid task', task instanceof Task);  
        verify('task is new', !this.#tasks.includes(task));  
        this.#tasks.push(task);  
    }  
  
    removeTask(taskToRemove) {  
        const taskIndex = this.#tasks.indexOf(taskToRemove);  
        verify('task is on board', taskIndex > -1);  
        this.#tasks.splice(taskIndex, 1);  
    }  
  
    getTasks(status = '') {  
        if (status === '') return this.#tasks.slice();  
        return this.#tasks.filter(task => task.status === status);  
    }  
}
```

The class `Task` captures the concept of an individual working item. Besides technical constraints, the component protects the logical invariants of the Domain Model. At all times, the status of a task may only contain a valid value. Also, an assignee must be set before transitioning to “in progress” and it cannot be removed during that state. The class `TaskBoard` implements the Entity for a task board, which is mainly a mutable collection of items. However, instead of identifiers, it encloses complete Entities. The operation `getTasks()` provides Entity access by returning either all tasks or the ones matching a given status.

This relationship design only makes sense when tasks are not considered to be standalone elements. Otherwise, identifier references should be preferred.

The last example of this subsection shows the usage of the Entities for the task board context ([run code](#)):

Task board context: Usage example

```
const assigneeId = generateId();
const taskBoard = new TaskBoard({id: generateId()});
const task = new Task({id: generateId(), title: 'write tests'});
taskBoard.addTask(task);
task.description = 'write unit tests for new feature';
task.assigneeId = assigneeId;
task.status = 'in progress';
console.log('tasks in progress', taskBoard.getTasks('in progress'));
console.log('attempting to unassign task');
task.assigneeId = undefined;
```

Task assignee cleanup

The Bounded Context implementations represent almost all aspects of their underlying Domain Models. However, one specified rule is not covered by the so-far provided source code: “When a member is removed from a team, all its assigned tasks need to be unassigned”. This requirement must be met without drawing a direct dependency from the team member component to the task board area. The integration of these individual parts can be done on the outside in a surrounding architectural layer. Note that the rule seemingly conflicts with the fact that in progress tasks need to be assigned. This constraint can be satisfied by changing the status of in progress items back to “todo” upon an assignee removal.

The final implementation provides a function to remove a team member and un-assign all its tasks ([run code](#)):

Task board context: Unassign abandoned tasks

```
const removeMemberAndUnassignTasks = (teamMemberId, taskBoard) => {
  const tasks = taskBoard.getTasks();
  const assignedTasks = tasks.filter(task => task.assigneeId === teamMemberId);
  assignedTasks.forEach(task => {
    if (task.status === 'in progress') task.status = 'todo';
    task.assigneeId = null;
  });
  team.removeMember(teamMemberId);
};

const teamMember = new TeamMember({
  id: generateId(), userId: generateId(), role: new Role('developer'),
});
const team = new Team({id: generateId()});
team.addMember(teamMember.id);

const taskBoard = new TaskBoard({id: generateId()});
const task = new Task({id: generateId(), title: 'write tests'});
taskBoard.addTask(task);
task.description = 'write unit tests for new feature';
task.assigneeId = teamMember.id;
task.status = 'in progress';

console.log(taskBoard);
removeMemberAndUnassignTasks(teamMember.id, taskBoard);
console.log(taskBoard);
```

The illustrated source code for the Sample Application represents a full implementation of all defined Domain Model parts. The enforcement of the overarching rule that affects multiple contexts is achieved with a separate surrounding functionality. While this approach works, its code design is not ideal. The integration of multiple independent parts can be improved through an event-based communication, which is covered in the next chapter.

Chapter 7: Domain Events

Domain Events represent something that happened in a Domain, for example when a user registers for a new account. [Evans, p. 20] describes them as “full-fledged part of the domain model”. This means, they should not only appear in the implementation but also in abstractions, concepts and visualizations. Their purpose is to convey meaningful knowledge about important specific occurrences. According to [Evans, p. 20], this should be something “that domain experts care about”. Therefore, Domain Events are not meant to contain purely technical information. The messages are used to facilitate communication, integration and consistency between separate software parts. In that sense, they support the correct functioning of a system. Nevertheless, the pattern is not to be interpreted as technical mechanism. The primarily conveyed information must always represent domain-specific knowledge.



Domain Events are facts

Domain Events may also be called **facts**. The usage of this term can have multiple advantages. For one, it clearly expresses that an event describes something that happened in the past. Also, it makes it more obvious that the occurrences can only be acknowledged, but cannot be rejected.

Relation to Event-driven Architecture

Event-Driven Architecture (EDA) is an architectural style, where event-based notifications are exchanged between multiple software parts. The primary goal is to establish loosely coupled messaging and therefore reduce and avoid direct dependencies. Furthermore, the approach can increase scalability, improves resilience and helps to make asynchronous processes more explicit. The concept itself is abstract and can be applied universally, independent of DDD, CQRS or Event Sourcing. For example, the Document Object Model (DOM) implemented in each browser software includes an event-based communication mechanism. Since Domain Events are typically distributed as notifications, their usage inherently constitutes a form of EDA. While this book does not discuss the architectural style specifically, its core ideas are applied in this and the following chapters.

Naming conventions

The type name for a Domain Event must be chosen carefully, as with every other part of a Domain Model. The selected term should unambiguously describe what happened. For that, it must be expressive and apply the respective Ubiquitous Language. One recommendation is to combine the affected subject and the executed action in past tense. Normally, the subject is the addressed component and the action is the invoked behavior. For example, the function `reload()` on a browser tab would produce the event type “BrowserTabReloaded”. Whenever possible, the subject should be put first. However, sometimes this leads to awkward names. For a newsletter function `subscribe()`, the event type would be “NewsletterSubscribed”. In such cases, the word order can be changed and additional terms may be included.

Exemplary Domain Event names

Subject	Action	Potential event type name
browser tab	reload()	BrowserTabReloaded
meeting	addParticipant()	MeetingParticipantAdded
guestbook	writeMessage()	GuestbookMessageWritten
user	login()	UserLoggedIn
newsletter	subscribe()	NewsletterSubscriptionAdded

Structure and content

Domain Events should only contain simple immutable attributes without functions. They must be easy to consume and commonly require to be serialized. Conceptually, an event consists of both data and metadata. The data part includes all specialized information. At a minimum, this is the type name of the event. Given that the affected subject has a conceptual identity, it should also be included. On top of that, any custom data can be contained. Typically, this incorporates all modified attributes and other related information. The exact structure should be adjusted to the needs of consumers. In contrast, the metadata part contains generic information independent of event types. Its most important value is a unique identifier for each event. Also, it often includes the time of occurrence.

Typical Domain Event contents

Information	Category	Description
type	data	Name of Domain Event Type
subject id	data	Identity of affected subject
custom data	data	Additional specialized data
id	metadata	Unique event identifier
creation time	metadata	Timestamp of event occurrence

What about a “timestamp” attribute?



Many reading materials and software utilities related to Domain Events define an attribute called “timestamp”. Typically, its contained value represents the time when an event was created. The issue is that the word “timestamp” is technical and merely describes the type of the enclosed information. In contrast, a more expressive term such as “creation time” conveys an actual meaning.

Event creation

There are multiple aspects to consider for the implementation of Domain Events. For one, their data structure should be homogenous across instances of the same type. In JavaScript, this requires a controlled creation mechanism, such as a constructor function. Secondly, a Domain Model implementation should only provide the specialized data, while the metadata is generated by another architectural layer. There are different solution approaches for this. Domain Model components can exclusively yield the specialized knowledge, which is later enriched with additional generic information. This keeps the Domain Layer free of unrelated concerns. Alternatively, the model components can make use of an event factory, which is additionally responsible for generating the metadata. The advantage of this approach is that the Domain Layer always produces complete events.

The following example provides an implementation of a factory for specialized Event types:

Events: Event Type Factory

```
let generateId = () => '';
let createMetadata = () => ({});

const createEventType = (type, dataStructure) => {
  const Event = function (data) {
    const containsInvalidFields = Object.keys(dataStructure).some(property => {
      const dataTypes = [].concat(dataStructure[property]);
      return dataTypes.every(dataType => typeof data[property] != dataType);
    });
    if (containsInvalidFields) throw new TypeError(
      `expected data structure: ${JSON.stringify(dataStructure, null, 2)}`
    );
    Object.assign(this,
      {type, data, id: generateId(), metadata: createMetadata()});
  };
  Event.type = type;
  return Event;
};

const setIdGenerator = idGenerator => generateId = idGenerator;

const setMetadataProvider = metadataProvider => createMetadata = metadataProvider;

const eventTypeFactory = {createEventType, setIdGenerator, setMetadataProvider};
```

The component `eventTypeFactory` enables to define event types with type-safe data and automatically generated metadata. Its operation `createEventType()` expects a type name and a data structure with property names and types. For polymorphic fields, a type array can be defined. The return value is an event constructor function. Upon its invocation, it validates given data against the previously provided structure. Type mismatches lead to an exception. In case of success, the given data is assigned. As event type name, the argument from the original factory call is used. Both the event identifier and the metadata are generated with the functions `generateId()` and `createMetadata()`. Their behavior is configurable through the operations `setIdGenerator()` and `setMetadataProvider()`. Every Event type exposes its type name as the static property `type`.



Domain Event structure

The event types produced via the type factory create events that do not strictly categorize all data and metadata. Both the event identifier and the type name are assigned to attributes on the root level. The reason is that this represents the most commonly used structure for Domain Events.

The illustrated factory component serves two important purposes. For one, it ensures structural consistency and type-safety of data across event instances of the same type. The implementation uses the `typeof` operator to primarily support primitive types, such as `boolean`, `number` and `string`. As Domain Events should only consist of simple attributes, the mechanism is sufficient. The second purpose is to free consumers from the infrastructural responsibility of creating metadata when instantiating events. This is achieved with an automatic and configurable generation of identifiers and additional metadata. Domain Model components can create full events without referring to specific infrastructural functionalities. Even more, the component `eventTypeFactory` itself is not tied to a specific implementation, as it only relies on configurable abstractions.

The next code shows an exemplary usage of the Event type factory ([run code](#)):

Events: Event Type Factory usage

```
const CommentWrittenEvent = eventTypeFactory.createEventType(
  'CommentWritten',
  {commentId: 'string', message: 'string', author: ['string', 'undefined']},
);

eventTypeFactory.setIdGenerator(generateId);
eventTypeFactory.setMetadataProvider(() => ({creationTime: new Date()}));

console.log(`type of event: ${CommentWrittenEvent.type}`);

console.log(new CommentWrittenEvent(
  {commentId: generateId(), message: 'Very nice content!'}));

console.log(new CommentWrittenEvent(
  {commentId: generateId(), message: 'Contact me!', author: 'john@example.com'}));

console.log(new CommentWrittenEvent({commentId: generateId()}));
```

The code starts with creating the specialized Domain Event type `CommentWrittenEvent` by invoking the factory `createEventType()`. The passed in data structure defines the attributes `contentId`, `comment` and `author`. While the content identifier and the comment are defined as `string`, the `author` attribute can be either a `string` or `undefined`. Next, the factory behavior is configured by executing the functions `setIdGenerator()` and `setMetadataProvider()`. For the identifier generation, the operation `generateId()` is used. For the additional metadata creation, an anonymous function is defined that creates an object containing a current timestamp. Finally, the event type `CommentWritten` is instantiated three times. The first two

executions succeed and return an event. In contrast, the third invocation throws an exception, as the passed in data contains an incorrect attribute type.



Default metadata creation

The code examples in this book follow a common approach for creating event metadata. The helper function `generateId()` is used as the standard mechanism for the identifier generation. Wherever needed, the additional metadata creation is handled by an operation that yields an object with the time of occurrence.

Distribution and processing

The main purpose of Domain Events is the distribution of knowledge about important occurrences. Potential recipients for this type of information are components within the same context, in foreign contexts and in external systems. On their own, Domain Events represent structured information about something that happened in a Domain. Making them available for further processing establishes a powerful communication exchange. Through the use of a suitable transport mechanism, the events can be consumed by any interested software part. This enables a far-reaching distribution of domain-related knowledge without tightly coupling producers and consumers. The processing may even happen asynchronously and in a deferred way. Independent of specific characteristics and employed technologies, establishing an event-based communication should have a minimal impact on a Domain Model implementation.



Events without distribution

While Domain Events are especially powerful in combination with their distribution, they can also be useful without it. For example, events on their own may act as change entries in a persistent audit log.

Publishing and Subscribing

There are different implementation approaches for the distribution of Domain Events. One is the application of the **Observer** design pattern. With this approach, a subject maintains a list of observers, which can be notified of events. The subject does not need to know the consumers, as they are only referenced through a common interface. In contrast, the

observers need to be aware of the concrete subject. Another more loosely coupled approach is the messaging pattern **Publish-Subscribe**. Here, an additional component called “Message Broker” or “Message Bus” is put in between publisher and subscriber. This frees both parts from the need to know each other. They only depend on the central messaging component. The pattern is especially useful for communication across processes or over the network.

The following example provides a basic implementation of a Message Bus component:

Message Bus: Basic implementation

```
class MessageBus {  
  
    #subscribersByTopic = new Map();  
  
    subscribe(topic, subscriber) {  
        const newSubscribers = this.#getSubscribers(topic).concat([subscriber]);  
        this.#subscribersByTopic.set(topic, newSubscribers);  
    }  
  
    unsubscribe(topic, subscriber) {  
        const subscribers = this.#getSubscribers(topic).slice();  
        subscribers.splice(subscribers.indexOf(subscriber), 1);  
        this.#subscribersByTopic.set(topic, subscribers);  
    }  
  
    publish(topic, message) {  
        this.#getSubscribers(topic).forEach(  
            subscriber => setTimeout(subscriber(message), 0));  
    }  
  
    #getSubscribers(topic) { return this.#subscribersByTopic.get(topic) || []; }  
}
```

This is accompanied by an exemplary usage of the component ([run code](#)):

Message Bus: Exemplary usage

```
const messageBus = new MessageBus();

messageBus.subscribe('topic-a', console.log);
messageBus.publish('topic-a', {data: 'something about a'});
messageBus.publish('topic-b', {data: 'something about b'});
```

The next example provides the implementation of a specialized Event Bus class:

Event Bus: Basic implementation

```
class EventBus {

    #messageBus;

    constructor(messageBus = new MessageBus()) {
        this.#messageBus = messageBus;
    }

    subscribe(eventType, subscriber) {
        return this.#messageBus.subscribe(eventType, subscriber);
    }

    unsubscribe(eventType, subscriber) {
        return this.#messageBus.unsubscribe(eventType, subscriber);
    }

    publish(event) {
        if (typeof event.type != 'string') throw new Error('invalid event');
        return this.#messageBus.publish(event.type, event);
    }
}
```

The class `MessageBus` enables to publish messages and to notify registered subscribers. Adding a callback function for a specific topic is done via the operation `subscribe()`. The command `publish()` is responsible for distributing a message. Upon execution, all registered callbacks for the given topic are invoked. This procedure is called “topic-based distribution”. The subscribers are executed asynchronously using `setTimeout()`. While the asynchronicity produces additional complexity, it prevents consumers from relying on a synchronous process. Removing a subscriber is done with the operation `unsubscribe()`. The component

EventBus represents a specialized distribution mechanism for Domain Events. As constructor argument, it accepts a Message Bus implementation and uses the class MessageBus by default. Its operation publish() expects an event object, whose type name is used as message topic.



Why not use EventEmitter?

The Node.js module events provides the class EventEmitter, which is effectively an implementation of the Publish-Subscribe pattern. Consequently, the component can also be used for Domain Event distribution. However, it does not support to await the completion of individual asynchronous subscriber functions as illustrated in the next subsection.

The last example of this subsection shows a usage of the Event Bus class ([run code](#)):

Event Bus: Exemplary usage

```
const eventBus = new EventBus();

const firstEvent = {
  type: 'UserSubscribedToMailingList', id: generateId(),
  data: {userId: generateId(), username: 'first user'},
  metadata: {creationTime: Date.now()},
};

const secondEvent = {
  type: 'UserSubscribedToMailingList', id: generateId(),
  data: {userId: generateId(), username: 'second user'},
  metadata: {creationTime: Date.now()},
};

const subscriber = event => console.log(`new subscriber: ${event.data.username}`);

eventBus.subscribe('UserSubscribedToMailingList', subscriber);
eventBus.publish(firstEvent).then(() => console.log('published first event'));
eventBus.publish(secondEvent).then(() => console.log('published first event'));
```

Guaranteed delivery

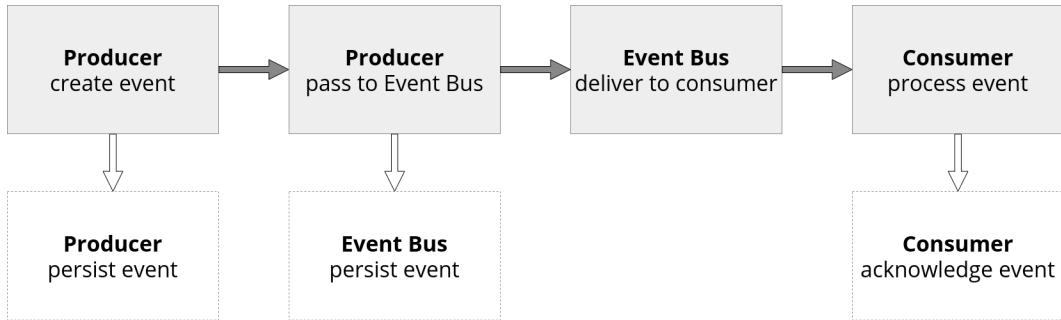


Figure 6.1: Domain Event distribution

The theory of Domain Events and their distribution is fairly trivial. When subscribers only execute secondary functionalities within the same process, the implementation is also simple. However, when a message notification triggers crucial actions or involves communication across processes, it gets more complex. The main challenge is the guaranteed distribution of events, which affects all parts involved in the communication. For one, the publisher must ensure that every event is correctly handed over to the bus. The Event Bus itself must guarantee the delivery to all registered subscribers. Finally, the subscribers must acknowledge each processed item. In most cases, this requires the use of persistent queues, potentially within all three parts. When done properly, the delivery can be guaranteed, even in case of software crashes.

The next code shows an extended implementation for the publishing operation of the Message Bus component that awaits asynchronous subscribers:

Message Bus: Publishing with async subscriber support

```

async publish(topic, message) {
    await Promise.all(this.#getSubscribers(topic).map(subscriber =>
        new Promise(resolve => setTimeout(() => {
            Promise.resolve(subscriber(message)).then(resolve);
        })),
    )));
}
  
```

The following example illustrates its usage with asynchronous event processing ([run code](#)):

Message Bus: Usage with asynchronous subscribers

```
const messageBus = new MessageBus();

messageBus.subscribe('topic', async message => {
  await new Promise(resolve => setTimeout(resolve, 15));
  console.log('message processed by async subscriber:', message);
});

messageBus.subscribe('topic', message => {
  console.log('message processed by sync subscriber:', message);
});

messageBus.publish('topic', {data: 'a'})
  .then(() => console.log('processing done'));
console.log('publishing issued');
```

The reworked function `publish()` of the component `MessageBus` enables to await the completion of both synchronous and asynchronous subscribers. Each callback execution is wrapped with a `Promise.resolve()` call to attach a success handler, even if the callback is synchronous. The enclosing `setTimeout()` invocation per subscriber is wrapped in a `Promise` instance that resolves after the callback completion. Finally, the resulting Promises are passed to the operation `Promise.all()` and their completion is awaited. The implementation makes it possible to wait until a published item is fully processed, even with asynchronous subscribers. This behavior enables to provide a guaranteed Domain Event delivery when combined with an according publishing queue. The usage code shows that the processing is only reported as done after the asynchronous subscriber operation completed.

Dealing with duplicates

One implication of guaranteed event delivery is that a single event can arrive multiple times at individual subscribers. This is because it is technically almost impossible to ensure that every item is delivered exactly once. Most distribution mechanisms provide an “at least once” guarantee. This means that an event is guaranteed to be delivered, but may arrive again in exceptional cases. One possibility for lowering the chances is by saving identifiers of consumed events and detecting duplicates directly before processing. However, subscriber operations must still be able to deal with duplicates. Whenever possible, their behavior should be idempotent. This means that processing an event repeatedly must have the same effect as when done once. This implementation style makes duplicates irrelevant, but is not always possible.

The following example provides a factory function for creating a subscriber with built-in event de-duplication ([run code usage](#)):

Domain Event: Factory for subscriber with de-duplication

```
const createSubscriberWithDeDuplication = originalSubscriber => {
  const processedEventIds = [];
  return event => {
    if (processedEventIds.includes(event.id)) {
      console.log(`dropping event duplicate with id: ${event.id}`);
      return;
    }
    originalSubscriber(event);
    processedEventIds.push(event.id);
  };
};
```

This is accompanied by an example of a subscriber to send order confirmation e-mails ([run code usage](#)):

Order confirmation: E-Mail sending with de-duplication

```
const OrderCreatedEvent = eventTypeFactory.createEventType('OrderCreated',
  {orderId: 'string', userId: 'string', email: 'string', cartItems: 'object'});

eventBus.subscribe(OrderCreatedEvent.type, createSubscriberWithDeDuplication(
  event => console.log(`send e-mail to ${event.data.email}`)));

const orderId = generateId(), userId = generateId(), email = 'john@example.com';
const event = new OrderCreatedEvent({orderId, userId, email, cartItems: /*...*/});

eventBus.publish(event);
eventBus.publish(event);
```

The factory `createSubscriberWithDeDuplication()` creates an operation that ignores event duplicates. As argument, it expects a subscriber function. The return value is another function that decorates the original behavior with a de-duplication mechanism. The identifiers of all processed events are saved in an array. Whenever a new event is received, its identifier is first searched in the array. In case of a duplicate, an error message is logged. Only if the event has not been processed yet, the original subscriber function is invoked. The order confirmation example illustrates an actual use case for de-duplication. For each “`OrderCreated`” event, an e-mail should be sent to the according user. With the help of the factory, every event is only processed once, independent of how often it is published.

Ordering of events

Events might be delivered out of the order they are published in. This can especially happen when a distribution spans across runtime processes or over the network. Ensuring a global ordering is possible, but requires major efforts within the publisher, the broker and the subscribers. In most cases, the benefits do not justify the associated complexity. An alternative is to exclusively ensure a linearized order within individual Entities. Still, this increases the overall complexity. The most commonly used setup is that events are typically delivered in order, but without any guarantee. When a subscriber requires a specific sequence, it must be established directly before processing. The Event Bus implementation of this chapter distributes events in order by relying on how the operation `setTimeout()` queues tasks.

Example: Classified ads platform

Consider operating an online classified ads platform to trade arbitrary goods. The business model is to provide paid services on top of a free version. This example focuses on a feature to watch for keywords in new ads and to notify users. The feature is introduced in two steps. First, custom lists are added as free functionality. This enables users to manually keep track of interesting offers. The second step is to provide an automatic list population of new ads matching a keyword together with e-mail notifications. This part is offered as paid service. As example, if a user is interested in chairs, the software can gather ads containing the keyword “chair”. Whenever an item is added to a list, the respective user gets notified.

Before developing the new functionality, one preliminary step is to review the existing Domain Model and its implementation. Overall, the source code of the Domain Layer consists of two individual parts. One is the Entity type for the classified ad concept. This component encloses a unique identifier, a title, a description and a price. Both the title and the description are represented as simple strings. Also, the title is constrained by the invariant that its length must not exceed 100 characters. In contrast, the description is not constrained and can therefore be of arbitrary length. The second model part is the price Value Object type, consisting of a value and a currency identifier.

The first examples show the code of the existing classified ad Entity and the price Value Object:

Classified ad platform: Classified ad Entity

```
class ClassifiedAd {  
  
    id; #title; description; price;  
  
    constructor({id, title, description, price}) {  
        Object.defineProperty(this, 'id', {value: id, writable: false});  
        Object.assign(this, {title, description, price});  
    }  
  
    get title() { return this.#title; }  
  
    set title(title) {  
        if (title.length > 100) throw new Error('max title length exceeded');  
        this.#title = title;  
    }  
}
```

Classified ad platform: Price Value Object

```
const Price = function({value, currency}) {  
    Object.freeze(Object.assign(this, {value, currency}));  
};
```

The next code provides an implementation of the previously described list component:

Classified ad platform: Classified Ad List

```
class ClassifiedAdList {  
  
    id; #classifiedAdIds = [];  
  
    constructor({id}) {  
        Object.defineProperty(this, 'id', {value: id, writable: false});  
    }  
  
    addClassifiedAd(classifiedAdId) {  
        this.#classifiedAdIds.push(classifiedAdId);  
    }  
  
    getClassifiedAds() { return this.#classifiedAdIds.slice(); }  
}
```

```
}
```

The component `ClassifiedAdList` expresses the Domain Model concept of a classified ad list as an Entity type. As its attributes, the class defines an identifier as well as an array of classified ad identities. Adding an ad to a list is done by passing its identifier to the operation `addClassifiedAd()`. The function `getclassifiedAds()` is responsible for returning the currently contained ones. The relationship between a list and its ads is expressed via identifier references as opposed to embedding complete Entities. This approach is feasible since the list component does not require further classified ad information to be locally available. With the additional Domain Model component, it is possible to create custom lists for manually tracking selected offers.

The first step towards an automatic list population is to introduce an event type for the creation of classified ads:

Classified ad platform: Classified ad creation event

```
const ClassifiedAdCreatedEvent = eventTypeFactory.createEventType(
  'ClassifiedAdCreated',
  {classifiedAdId: 'string', title: 'string', description: 'string',
   priceValue: 'number', priceCurrency: 'string'},
);
```

The next code provides an extended version of the classified ad Entity that publishes the new event:

Classified ad platform: Classified Ad Entity with event

```
class ClassifiedAd {

  id; #title; description; price;

  constructor({id, title, description, price, eventBus}) {
    Object.defineProperty(this, 'id', {value: id, writable: false});
    Object.assign(this, {title, description, price});
    eventBus.publish(new ClassifiedAdCreatedEvent(
      {classifiedAdId: this.id, title, description,
       priceValue: price.value, priceCurrency: price.currency}));
  }

  get title() { return this.#title; }
```

```
    set title(title) {
        if (title.length > 100) throw new Error('max title length exceeded');
        this.#title = title;
    }
}
```

The following usage demonstrates a subscription for the classified ad creation event ([run code](#)):

Classified ad platform: Classified Ad Entity with event usage

```
const eventBus = new EventBus();
eventBus.subscribe(ClassifiedAdCreatedEvent.type, console.log);

const price = new Price({value: 40, currency: 'EUR'});
new ClassifiedAd({id: generateId(),
    title: 'Unused chair', description: 'Retro look', price, eventBus});
```

The Domain Event type `ClassifiedAdCreatedEvent` is created with the factory function `createEventType()`. Its data structure encloses all attributes for the initial state of an ad Entity. The constructor of the class `ClassifiedAd` is extended to create and publish an according event instance. For this purpose, it expects an Event Bus as additional constructor argument. The publishing happens only after the invariant protection and the initial state assignment. Consequently, the event only occurs for successfully created ads, but not for failed attempts. The exemplary usage demonstrates how to subscribe to the Domain Event. For each created Entity instance, the function `console.log()` is executed. This again illustrates the loose coupling between publishers and subscribers. Both only need to know the central messaging component, but not each other.

The next code implements a service component for determining whether a keyword is contained in a given text:

Classified ad watcher: Keyword service

```
const keywordService = {
  containsKeyword: (text, word) => text.toLowerCase().includes(word.toLowerCase())),
};
```

The service is used for a function that enables an automatic classified ad list population:

Classified ad watcher: Classified ad list population

```
const activateClassifiedAdListPopulation = ({eventBus, list, keyword}) =>
  eventBus.subscribe(ClassifiedAdCreatedEvent.type, ({data}) => {
    const {classifiedAdId, title, description} = data;
    const containsKeyword = [title, description].some(
      text => keywordService.containsKeyword(text, keyword));
    if (containsKeyword) {
      list.addClassifiedAd(classifiedAdId);
      console.log('new classified ad with matching keyword:', classifiedAdId);
    }
  });
};
```

The usage of the functionality can be seen in the next example ([run code](#)):

Classified ad watcher: Classified ad list population usage

```
const eventBus = new EventBus();

const list = new ClassifiedAdList({eventBus, id: generateId()});
activateClassifiedAdListPopulation({eventBus, list, keyword: 'chair'});

new ClassifiedAd({
  id: generateId(), title: 'Unused chair', description: 'Office chair',
  price: new Price({value: 40, currency: 'EUR'}), eventBus,
});
new ClassifiedAd({
  id: generateId(), title: 'Used laptop', description: 'Lenovo T400',
  price: new Price({value: 200, currency: 'EUR'}), eventBus,
});
```

The object `keywordService` represents a Domain Service to determine whether a keyword is contained in a text. Its operation `containsKeyword()` expects a text and a keyword and returns `true` if the term is found. The operation `activateClassifiedAdListPopulation()` is

responsible for registering a Domain Event subscriber that adds classified ads to a list that contain a given term. As arguments, it expects an Event Bus, a classified ad list and a keyword. The function subscribes to the event type “ClassifiedAdCreated”. For each occurrence, the callback uses the operation `containsKeyword()` to check whether the given keyword is contained in the new ad. If the term is found, the classified ad identifier is added to the list.

The following code defines another Domain Event type, which represents the addition of a classified ad to a list:

Classified ad platform: Classified ad list addition event

```
const ClassifiedAdAddedToListEvent = eventTypeFactory.createEventType(
  'ClassifiedAdAddedToListEvent',
  {classifiedAdListId: 'string', classifiedAdId: 'string'},
);
```

This is followed by an extended implementation of the list component, which publishes the newly introduced event:

Classified ad platform: Classified ad list Entity with event

```
class ClassifiedAdList {

  id; #classifiedAdIds = []; #eventBus;

  constructor({id, eventBus}) {
    Object.defineProperty(this, 'id', {value: id, writable: false});
    this.#eventBus = eventBus;
  }

  addClassifiedAd(classifiedAdId) {
    this.#classifiedAdIds.push(classifiedAdId);
    this.#eventBus.publish(new ClassifiedAdAddedToListEvent(
      {classifiedAdListId: this.id, classifiedAdId}));
  }

  getClassifiedAds() { return this.#classifiedAdIds.slice(); }

}
```

The next example provides an operation for setting up a notification mechanism of classified ad list additions:

Classified ad platform: Classified ad list notification

```
const setupClassifiedAdListNotification = ({eventBus, list}) =>
  eventBus.subscribe(ClassifiedAdAddedToListEvent.type, event => {
    if (event.data.classifiedAdListId === list.id)
      console.log(`sending e-mail for classified ${event.data.classifiedAdId}`);
  });

```

The final code shows an exemplary usage of the classified ad list notification mechanism ([run code usage](#))

Classified ad platform: Classified ad list notification usage

```
const eventBus = new EventBus();

const list = new ClassifiedAdList({eventBus, id: generateId()});
activateAutomaticListPopulation({eventBus, list, keyword: 'chair'});
setupClassifiedAdListNotification({eventBus, list});

new ClassifiedAd({
  id: generateId(), title: 'Unused chair', description: 'Office chair',
  price: new Price({value: 40, currency: 'EUR'}), eventBus,
});
new ClassifiedAd({
  id: generateId(), title: 'Used laptop', description: 'Lenovo T400',
  price: new Price({value: 200, currency: 'EUR'}), eventBus,
});
```

The Domain Event type `ClassifiedAdAddedToListEvent` represents the addition of a classified ad to a list. The class `ClassifiedAdList` is extended to create and publish an event of this type when an identifier is added. This is required for the notification mechanism. Similar to the implementation in the classified ad component, the publishing happens only after the state modification. The operation `setupClassifiedAdListNotification()` registers an event subscriber that notifies of identifier additions for a specific list. As arguments, it expects an Event Bus and a classified ad list. Its callback operation checks whether a received event is about the passed in list and conditionally outputs a placeholder notification. Apart from the exemplary notification, the illustrated Domain Model implementation provides the full functionality as specified in the beginning.

Overall, this example illustrates how to utilize Domain Events for a loosely coupled information exchange and for enabling reactive behavior.

Sample Application: Event-based integration

This section illustrates the introduction of Domain Events for the Sample Application. The motivation is to enable a loosely coupled integration between components of the project part and the task board part. For this purpose, a single Domain Event type is sufficient. As a consequence, this section only takes a glimpse at Domain Events and their distribution. Nevertheless, the full implementation of the Sample Application requires multiple event types and various integration points. Even more, both the application of CQRS and Event Sourcing built upon the use of Domain Events. Therefore, the introduction of this pattern and its associated generic functionalities lay the foundation for subsequent implementation steps.

Shared functionalities

The use of Domain Events requires to add three generic functionalities to the source code of the Sample Application. One part is the component `eventTypeFactory` for creating type-safe specialized event types with automatically generated metadata. For the distribution of Domain Events, the Event Bus component is re-used, which includes the underlying Message Bus component. The event type factory is placed in the shared Domain part, as it is agnostic of specific infrastructural aspects. While the Message Bus and the Event Bus are also free of technological details, they are placed in the shared Infrastructure part. Consider the scenario when replacing their in-memory mechanics with a persistent implementation. In this case, it would be obvious that the components architecturally belong to the Infrastructure Layer.

Implementation changes

The source code from the last chapter represents a complete expression of the Domain Model. However, the code design of the implementation to unassign tasks upon a team member removal is questionable. This is because it introduces an additional component around the task board and the team Entity types. These two parts belong to the Domain Layers of different context implementations. In fact, the approach introduces another conceptual boundary that depends on the two areas. The introduction of Domain Events enables to mitigate this design issue. Through the facilitation of a loosely coupled message exchange, it is possible to remove the additional layer. Instead, the two context implementations depend on a central messaging component and a common message format.

The first code provides the definition of a Domain Event type to represent a team member removal:

Project context: Team member removal event

```
const TeamMemberRemovedFromTeamEvent = createEventType(
  'TeamMemberRemovedFromTeam', {teamId: 'string', teamMemberId: 'string'});
```

The next implementation shows a reworked team component that publishes an event whenever a member is removed:

Project context: Team Entity with event

```
class Team {

  id; #teamMemberIds = []; #eventBus;

  constructor({id, eventBus}) {
    verify('valid id', id != null);
    Object.defineProperty(this, 'id', {value: id, writable: false});
    this.#eventBus = eventBus;
  }

  addMember(teamMemberId) {
    verify('team member is new', !this.#teamMemberIds.includes(teamMemberId));
    this.#teamMemberIds.push(teamMemberId);
  }

  removeMember(teamMemberIdToRemove) {
    const indexToRemove = this.#teamMemberIds.indexOf(teamMemberIdToRemove);
    if (indexToRemove === -1) return;
    this.#teamMemberIds.splice(indexToRemove, 1);
    this.#eventBus.publish(new TeamMemberRemovedFromTeamEvent(
      {teamId: this.id, teamMemberId: teamMemberIdToRemove}));
  }

  get teamMemberIds() { return this.#teamMemberIds.slice(); }
}
```

The third example introduces a component that synchronizes affected tasks on a board upon a team member removal:

Task board context: Task board synchronization

```
const taskBoardSynchronization = {

  activateTaskAssigneeSynchronization({eventBus, taskBoard}) {
    eventBus.subscribe('TeamMemberRemovedFromTeam', ({data: {teamMemberId}}) => {
      const tasks = taskBoard.getTasks();
      const assignedTasks = tasks.filter(task => task.assigneeId === teamMemberId);
      assignedTasks.forEach(task => {
        if (task.status === 'in progress') task.status = 'todo';
        task.assigneeId = undefined;
      });
    });
  },
};

};


```

The reworked class `Team` expects an Event Bus as additional constructor argument. Executing its command `removeMember()` causes to publish a “`TeamMemberRemoved`” event after the state change. The event data consists of the team identity and the identifier of the removed member. Although the message is required for a technical reason, it represents meaningful Domain knowledge. The component `taskBoardSynchronization` defines the operation `activateTaskAssigneeSynchronization()` for un-assigning tasks from a team member when it is removed. As arguments, the function expects an Event Bus and a task board. The subscriber callback first retrieves the tasks of the given task board. For each item, it checks whether the removed team member is the assignee. In that case, the assignee is removed and the status is conditionally changed to “`todo`”.

The final example demonstrates the automatic un-assignment of tasks upon a team member removal without an additional surrounding component ([run code](#)):

Sample Application: Overall usage

```
const eventBus = new EventBus();

const teamMember = new TeamMember(
  {id: generateId(), userId: generateId(), role: new Role('developer')});
const team = new Team({id: generateId(), eventBus});
team.addMember(teamMember.id);

const taskBoard = new TaskBoard({id: generateId()});
taskBoardSynchronization.activateTaskAssigneeSynchronization({eventBus, taskBoard});
```

```
const task = new Task({id: generateId(), title: 'write tests'});
taskBoard.addTask(task);
task.description = 'write unit tests for new feature';
task.assigneeId = teamMember.id;
task.status = 'in progress';

console.log(`assignee: ${task.assigneeId}, status: ${task.status}`);
team.removeMember(teamMember.id);
setTimeout(() =>
  console.log(`assignee: ${task.assigneeId}, status: ${task.status}`), 0);
```

For the code design of the Domain Model implementation, the event-based integration of separate contexts is an important improvement. The next step is to identify and implement appropriate consistency boundaries. This is an essential stepping stone towards a concurrent and persistent Domain Model implementation.

Chapter 8: Aggregates

An Aggregate defines a transactional consistency boundary that encloses a collection of related Domain Model components. [Evans, p. 126] describes it as “cluster of associated objects that we treat as unit for the purpose of data changes”. This means that within such a boundary each individual change must transactionally result in a consistent state. Aggregates are essential for concurrent component access, which typically occurs in combination with persistence. The concept itself does not require the use of specific technical constructs, but influences the overall Domain Model implementation. [Vernon, p. 355] explains that Aggregates are “consistency boundaries and not driven by a desire to design object graphs”. Still, their use is not necessarily weakening the expression of Domain knowledge. Rather, it is embracing transactions and consistency as important aspects.

Transactions

A **transaction** encloses a series of operations to be treated as a single unit of work. This makes it possible to combine multiple steps into a logically inseparable procedure. Typically, this mechanism is used for changes to persistent data. With regard to databases, transactions must have four characteristics: atomicity, consistency, isolation and durability (ACID). Atomicity means that either all changes are applied or none of them. Consistency enforces defined invariants to be protected. Isolation ensures that concurrent operations produce the same outcome as if they were executed sequentially. Durability demands that the result of a transaction is non-volatile. Except for durability, these aspects can be interpreted as general requirements for transactional boundaries, independent of databases. Consequently, Aggregates should also adhere to these principles.

1 Breakdown of ACID principles

2 

3  Atomicity: apply all changes or none

Consistency: protect invariants

Isolation: eliminate concurrency conflicts

Durability: persist changes reliably

Structure and access

An Aggregate can have an arbitrary structure. The only constraint is to contain at least one Entity. This is because there must always be a designated root element. The **Aggregate Root** represents an entry point and is the only element to directly access or reference from the outside. Apart from that, any combination of Entities and Value Objects can be enclosed. The contained elements may even only have a local identity. The Aggregate Root should expose facade functions for a controlled access to its subordinate parts. This design makes it possible to ensure atomicity and consistency of related operations and therefore provide transactional behavior. Without such a boundary, it is impossible to guarantee that multiple individual state alterations are treated as single unit.



Implicit Aggregates

Every Entity that is not considered a part of another Aggregate, is implicitly an Aggregate on its own.

Example: Message inbox

Consider implementing the Domain Model for a message inbox. This type of functionality can be used on its own or as part of another software. The relevant model part for the example consists of two components. One is the inbox, which is a mutable collection of messages, to which new items can be added. The second component is the message itself, which contains content and a read status. Each message can be individually marked as read or as unread. On top of that, it must be possible to mark all items in an inbox as read at once. Both components must be identifiable independent of their state. For this example, the challenge is to create an adequate code design with regard to transactions and consistency.

The first code shows the implementation of the message Entity component:

Message inbox: Message Entity

```
class Message {  
  
    id; content; #isRead = false;  
  
    constructor({id, content}) {  
        Object.defineProperty(this, 'id', {value: id, writable: false});  
        Object.defineProperty(this, 'content', {value: content, writable: false});  
    }  
  
    markAsRead() { this.#isRead = true; }  
  
    markAsUnread() { this.#isRead = false; }  
  
    get isRead() { return this.#isRead; }  
  
}
```

The class `Message` expresses the concept of an individual message. This component can be used on its own without surrounding structure. Independent of state, it is possible to distinguish multiple instances through their conceptual identity. The public attribute `content` contains the actual message content. Changing the read status is done via the commands `markAsRead()` and `markAsUnread()`. Retrieving the current status is achieved through the operation `get isRead()`. The Entity type on its own already expresses essential Domain Model behavior. One implementation enhancement would be to prevent a message from transitioning into a read state it is already in. However, this is more a matter of validation and not of invariant protection. Marking a message as read or unread multiple times consecutively does not violate any constraint.

The next implementation shows the first approach for an inbox component ([run code usage](#)):

Message inbox: Inbox with identifiers

```
class Inbox {  
  id; #messageIds = [];  
  
  constructor({id}) {  
    Object.defineProperty(this, 'id', {value: id, writable: false});  
  }  
  
  addNewMessage(messageId) {  
    this.#messageIds.push(messageId);  
  }  
  
  getMessageIds() { return this.#messageIds.slice(); }  
}
```

Message inbox: Usage with identifiers

```
const first messageId = generateId(), second messageId = generateId();  
  
const messageDatabase = new Map();  
messageDatabase.set(first messageId, new Message(  
  {id: first messageId, content: 'Oranges are cheap today!'}));  
messageDatabase.set(second messageId, new Message(  
  {id: second messageId, content: 'Buy two apples, get one free!'}));  
  
const inbox = new Inbox({id: generateId()});  
inbox.addNewMessage(first messageId);  
inbox.addNewMessage(second messageId);  
  
inbox.getMessageIds().forEach(  
  messageId => messageDatabase.get(messageId).markAsRead());  
  
console.log(Array.from(  
  messageDatabase.values()).map(({content, isRead}) => ({content, isRead})));
```

The class `Inbox` is mainly a registry for message Entities. Invoking its command `addNewMessage()` adds an identifier. The query `getMessageIds()` retrieves the currently contained ones. Marking all messages as read at once is achieved by determining all registered Entities and marking them individually. This process is demonstrated by the usage code. The relation

between identifiers and Entities is resolved through an in-memory database. Although the implementation fulfills the requirements, the mechanism for marking all messages is problematic. This is because the specialized behavior resides outside a model component. Consequently, it is impossible for the inbox to ensure the atomicity and consistency of this process. Rather, every consumer is responsible for correctly carrying out the operations. This issue expresses the need for a larger enclosing Aggregate.

The second approach shows the implementation of an enclosing Aggregate with the inbox as root component ([run code usage](#)):

Message inbox: Inbox with Entities

```
class Inbox {  
  
    id; #messages = [];  
  
    constructor({id}) {  
        Object.defineProperty(this, 'id', {value: id, writable: false});  
    }  
  
    addNewMessage(id, content) {  
        this.#messages.push(new Message({id, content}));  
    }  
  
    markMessageAsRead(id) {  
        this.#messages.find(message => message.id === id).markAsRead();  
    }  
  
    markMessageAsUnread(id) {  
        this.#messages.find(message => message.id === id).markAsUnread();  
    }  
  
    markAllMessagesAsRead() {  
        this.#messages.forEach(message => message.markAsRead());  
    }  
}
```

Message inbox: Usage with enclosing Aggregate

```
const first messageId = generateId(), second messageId = generateId();

const inbox = new Inbox({id: generateId()});
inbox.addNewMessage(first messageId, 'Oranges are cheap today!');
inbox.addNewMessage(second messageId, 'Buy two apples, get one free!');

inbox.markAllMessagesAsRead();
```

The revised implementation of the class `Inbox` encloses complete message Entities and acts as single entry point for all behavior. Instead of referencing identifiers of existing elements, new Entities are created and registered via the command `addNewMessage()`. This design ensures identification capabilities from the outside and at the same time protects contained Entities from arbitrary access. Marking a message as read or unread is done via the facade functions `markMessageAsRead()` and `markMessageAsUnread()`. The Domain Model concept of marking all messages at once is expressed through the command `markAllMessagesAsRead()`. Since the Entities are contained, there is no need for an identifier-based lookup. The enclosing Aggregate can guarantee the atomicity and consistency of this compound operation. Furthermore, the code for creating, adding and marking messages is simplified.

Concurrency

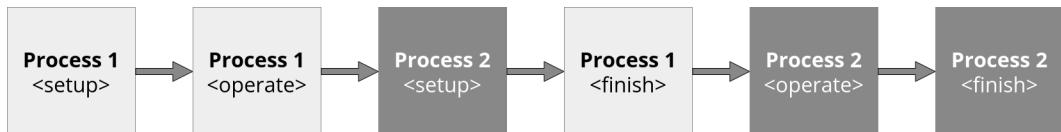


Figure 7.1: Concurrency example

Concurrency allows multiple tasks to be executed in overlapping time periods. This makes it possible for one computation to start running, even though another one is still in progress. There are various advantages to this mechanism. Physical resources can be used efficiently, program throughput is increased and responsiveness is possible, even when executing long-running operations. Typically, concurrent computing requires some form of threading. By default, JavaScript runtimes execute all given code within one main thread. This limitation can be eliminated by using worker threads. However, concurrency is also possible within a single main thread by making use of asynchronous and non-blocking programming

techniques. For example, splitting up the steps of a computation with timeouts allows for another series of operations to start in between.



Relation to parallelism

Parallelism allows multiple computations to be executed at the same time. This is possible through the use of multiple processing units. While the two concepts partially overlap, they must also be clearly differentiated. Within the context of this book, the considerations on concurrency apply almost equally to parallelism.

Example: Prime number test

Consider implementing a Domain Service to determine whether a given value is a prime number. According to the definition, this is true for every number larger than 1 that can only be divided by itself. One pragmatic approach for the implementation of this behavior is to loop through all possible divisors. For larger values, this calculation can take a notable amount of time. Using a synchronous approach, the main thread is completely blocked and unresponsive until the operation is finished. Yielding the process throughout the computation with asynchronous utilities, such as `setTimeout()`, enables the software to remain responsive. Even more, it enables to execute multiple tests concurrently, or to report the progress of a long-running operation.

The first example shows a simple synchronous implementation ([run code](#)):

Prime number test: Synchronous

```
const isPrimeNumber = number => {
  if (number <= 1) return false;
  for (let divisor = 2; divisor < number; divisor++)
    if (number % divisor === 0) return false;
  return true;
};

const testWithTiming = number => {
  console.time(`duration for ${number}`);
  console.log(`is ${number} prime number: ${isPrimeNumber(number)}`);
  console.timeEnd(`duration for ${number}`);
};

testWithTiming(524287);
testWithTiming(5);
```

The next code provides an asynchronous and non-blocking approach ([run code](#)):

Prime number test: Asynchronous

```
const isPrimeNumber = (number, config = {loopChunkSize: 500}) => {
  if (number <= 1) return Promise.resolve(false);
  let divisor = 2;
  const checkNextChunkRecursively = resolve => {
    const nextDivisorLimit = Math.min(divisor + config.loopChunkSize, number);
    for (divisor; divisor < nextDivisorLimit; divisor++)
      if (number % divisor === 0) return resolve(false);
    if (divisor === number) resolve(true);
    else setTimeout(() => checkNextChunkRecursively(resolve), 0);
  };
  return new Promise(checkNextChunkRecursively);
};

const testWithTiming = async number => {
  console.time(`duration for ${number}`);
  const result = await isPrimeNumber(number);
  console.log(`is ${number} prime number: ${result}`);
  console.timeEnd(`duration for ${number}`);
};

testWithTiming(524287);
testWithTiming(5);
```

Both implementations define the operation `isPrimeNumber()`. The synchronous version uses a simple for-loop and returns a boolean, indicating whether a given value is a prime number. The second variant executes multiple smaller loops asynchronously using `setTimeout()`, and returns a Promise that resolves with the resulting flag. The usage of the first implementation shows that multiple requests execute sequentially. In this case, the subsequent simpler test must wait for the first and more complex one to be finished. In contrast, the asynchronous variant allows the less complex computation to finish first, regardless of the order. Note that the overall execution time is significantly increased with the second approach. This is because of the overhead associated with the utilization of the JavaScript event loop.

Motivation and causes

The primary motivation for enabling concurrency is an increase in performance. For Node.js, asynchronous and non-blocking programming techniques allow to maximize the main

thread's efficiency. Regardless of the motivation, there are different causes for concurrency. One is the previously illustrated asynchronicity of a computation. Although this characteristic is normally the result of a conscious decision, its implications are sometimes not considered fully. Still, the more common cause for concurrency is the use of persistence. This can be again sub-divided into two scenarios. One is that loading and saving data is typically asynchronous itself, making it similar to the first cause. The second scenario is when persistent data is accessible in different programs. Independent of the cause, concurrency requires strategies to prevent conflicts and inconsistencies.

Concurrency control

Concurrency control ensures that concurrently executed tasks produce correct results. This is especially important for procedures that work on a shared resource. One fundamental requirement is that every involved operation receives an isolated representation of a resource. Sharing common data structures risks inconsistencies and consequential errors. As an example, assume that two people collaborate on a single shopping list. One of them deletes an entry, while the other adds an item. Both then check the updated list length to verify their action. However, the value effectively remains unchanged and may cause confusion. When they instead work on separate representations, their local changes do not affect each other. Persistent storage mechanisms often implicitly provide this isolation, as consumers access replicated data instead of actual resources.

Another important aspect is that persistent changes must not overwrite each other. Allowing modification requests regardless of their actuality can cause corruption and data loss. This problem arises when multiple transactions concurrently alter an isolated resource representation and save it. Reconsider the previous shopping list example. When the two persons both simultaneously load the list, make changes and save them, one of the transactions overwrites the other. Generally, modifications should only be accepted when they are based on the latest version. One solution approach is to make resource access exclusive. This prevents overwrites altogether, but it decreases performance. Alternatively, the actuality of changes can be verified upon saving. Outdated requests are rejected and must be retried. This produces a better performance but increases the complexity.

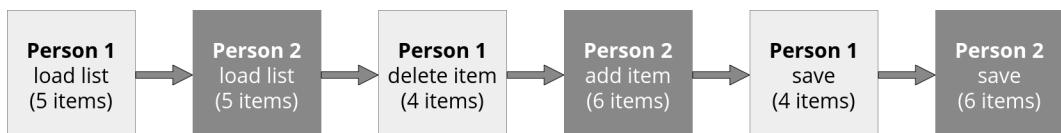


Figure 7.2: Overwrite changes

Example: Shopping list

This subsection illustrates the previously described challenges of concurrent component access and provides an exemplary solution approach. The shopping list scenario that was introduced earlier is re-used as example topic. This means the relevant Domain Model exclusively defines the shopping list component, which is a mutable collection of shopping items. The required behavior consists of creating a list, adding and removing an entry and querying the current list length. Individual items are represented as plain immutable string values. On top of the domain-specific implementation, the subsection provides a simple persistence mechanism and a concurrency utility.

The first example shows the implementation of the shopping list Entity component ([run code usage](#)):

Shopping list: Shopping list Entity

```
class ShoppingList {  
  
    id; #items;  
  
    constructor({id, items = []}) {  
        Object.defineProperty(this, 'id', {value: id, writable: false});  
        Object.defineProperty(  
            this, 'items', {get: () => this.#items.slice(), enumerable: true});  
        this.#items = items;  
    }  
  
    addItem(item) {  
        this.#items.push(item);  
    }  
  
    removeItem(item) {  
        const index = this.#items.indexOf(item);  
        if (index > -1) this.#items.splice(index, 1);  
    }  
  
    getLength() { return this.#items.length; }  
}
```

The class `ShoppingList` expresses the Domain Model concept of a shopping list as Entity type. Its attributes consist of a conceptual identity and a list of shopping items. Adding an

item to a list is done with the command `addItem()`. Read access is provided via a getter function. This property accessor also serves a persistence-specific purpose. Shopping list Entities should be serialized in a way they can easily be reconstructed again. For this reason, the constructor defines the getter with the enumerable flag set to true. The motivation is that the operation `JSON.stringify()` only includes enumerable attributes. This implementation approach enables to serialize an Entity as JSON string without explicit conversion code. Note that the approach is only used for this particular example.

The next implementation demonstrates two transactions that share a single data structure ([run code](#)):

Shopping list: Shared data structure

```
const shoppingList = new ShoppingList({id: generateId()});

const transaction1 = [
  () => console.log(`T1: accessing list with ${shoppingList.getLength()} items`),
  () => { console.log('T1: add "apples"'); shoppingList.addItem('apples'); },
  () => console.log(`T1: verify list with ${shoppingList.getLength()} items`),
];
const transaction2 = [
  () => console.log(`T2: accessing list with ${shoppingList.getLength()} items`),
  () => { console.log('T2: add "oranges"'); shoppingList.addItem('oranges'); },
  () => console.log(`T2: verify list with ${shoppingList.getLength()} items`),
];

const mergeTransactions = (transaction1, transaction2) =>
  transaction1.flatMap((operation, index) => [operation, transaction2[index]]);

mergeTransactions(transaction1, transaction2).forEach(operation => operation());
//transaction1.concat(transaction2).forEach(operation => operation());
```

The code starts with creating a shopping list Entity. Next, it defines two arrays of operations, of which each represents a separate transaction. Both arrays contain three operations. First, the shared list is accessed and its initial state is output. Secondly, a modification is announced and the according command is executed. As last step, the altered state is logged to verify the transactional success. For a concurrent execution, the helper function `mergeTransactions()` is implemented. When executing the code, the output indicates consistency failures due to incorrect verification messages. Each transaction only adds a single item, but ends up with a list length incremented by two. The commented code can be used for a sequential execution. This consistency failure demonstrates that concurrent access requires data isolation.

As preparation for the next use cases, the following example provides a file storage component as persistence mechanism ([run code usage](#)):

Filesystem: JSON File storage

```
class JSONFileStorage {  
  
    #storageDirectory;  
  
    constructor(storageDirectory) {  
        fs.mkdirSync(storageDirectory, {recursive: true});  
        this.#storageDirectory = storageDirectory;  
    }  
  
    async save(id, data) {  
        const filePath = `${this.#storageDirectory}/${id}.json`;  
        const tempFilePath = `${filePath}-${generateId()}.data`;  
        await writeFile(tempFilePath, JSON.stringify(data));  
        await rename(tempFilePath, filePath);  
    }  
  
    load(identifier) {  
        const filePath = `${this.#storageDirectory}/${identifier}.json`;  
        return readFile(filePath, 'utf-8').then(JSON.parse);  
    }  
}
```

The class `JSONFileStorage` combines JSON serialization with a filesystem-based storage. Its constructor expects a storage directory and ensures its existence by executing the function `mkdirSync()`. Saving an Entity is done with the command `save()`. The operation expects an identity and a data structure. First, it determines the file path by combining the storage directory and the given identifier. Then, it creates a temporary file path with a random identifier created by the operation `generateId()`. Next, the data is serialized as JSON string and written to the temporary file. Afterwards, the file is renamed to the target filename. Loading a persisted Entity is achieved with the operation `load()`. This function constructs a file path with the given identifier, loads the file content and de-serializes the retrieved string.



Atomicity of filesystem operations

Concurrently writing to the same file using `fs.writeFile()` can produce corrupted data. This is because the operation is not guaranteed to be atomic. In contrast, `fs.rename()` can only either succeed or fail. Therefore, writing to a temporary file and renaming it afterwards prevents data corruption due to concurrent modification.

The next code illustrates a change overwrite due to isolated data manipulation of a concurrently accessed component ([run code](#)):

Shopping list: Change overwrite

```
const shoppingListStorage = new JSONFileStorage(storageDirectory);

const listId = generateId();

const addShoppingListItem = async item => {
    const shoppingList = new ShoppingList(await shoppingListStorage.load(listId));
    shoppingList.addItem(item);
    await shoppingListStorage.save(listId, shoppingList);
};

const shoppingList = new ShoppingList({id: generateId()});
await shoppingListStorage.save(listId, shoppingList);
await Promise.all([addShoppingListItem('apples'), addShoppingListItem('oranges')]);
const savedShoppingList = await shoppingListStorage.load(listId);
console.log('items after both transactions:', savedShoppingList.items);
```

First, the code creates an instance of the file storage component `JSONFileStorage`. This is followed by the definition of the helper function `addShoppingListItem()`. The command loads an isolated resource representation, adds a given string as new item, and saves the modified state. Next, an exemplary Entity is created and persisted. As actual use case, two transactions to add a new item are executed concurrently. In this case, the concurrency is caused by the asynchronicity of the persistence mechanism combined with the sequential transaction initiation. Regardless of the specific execution order, this causes both operation sets to overlap in some way. As last step, the Entity is loaded again to output its state. Executing the code shows that only one of the items is successfully added.

As preparation for the final use case, the following example provides a helper component for exclusive resource access ([run code usage](#)):

Shopping list: Exclusive access

```
class ExclusiveAccess {  
  
    #isAccessLocked = false; #pendingPromiseResolves = [];  
  
    requestAccess() {  
        return new Promise(resolve => {  
            if (!this.#isAccessLocked) {  
                this.#isAccessLocked = true;  
                resolve();  
            } else this.#pendingPromiseResolves.push(resolve);  
        });  
    }  
  
    releaseAccess() {  
        if (this.#pendingPromiseResolves.length > 0)  
            setTimeout(this.#pendingPromiseResolves.shift(), 0);  
        else this.#isAccessLocked = false;  
    }  
}
```

The class `ExclusiveAccess` is a utility to provide a simple locking mechanism. One instance can serve as concurrency control for a specific resource. The function `requestAccess()` requests exclusive access and returns a `Promise` object, which resolves when the access is granted. There are two possible behaviors for this command. When there is currently no lock, it immediately resolves and prevents further access. Otherwise, it enqueues the request. The command `releaseAccess()` releases the current lock. This operation also behaves in two different ways. When there are no enqueued requests, the lock is released completely. Otherwise, the next pending access `Promise` is resolved. The component makes it possible for consumers to wait for the availability of a resource and to gain exclusive access.

The final use case shows an implementation with controlled concurrency through exclusive access ([run code](#)):

Shopping list: Concurrency control through exclusive access

```
const shoppingListStorage = new JSONFileStorage(storageDirectory);
const shoppingListAccess = new ExclusiveAccess();

const listId = generateId();

const addShoppingListItem = async item => {
    await shoppingListAccess.requestAccess();
    const shoppingList = new ShoppingList(await shoppingListStorage.load(listId));
    shoppingList.addItem(item);
    await shoppingListStorage.save(listId, shoppingList);
    shoppingListAccess.releaseAccess();
};

const shoppingList = new ShoppingList({id: generateId()});
await shoppingListStorage.save(listId, shoppingList);
await Promise.all([addShoppingListItem('apples'), addShoppingListItem('oranges')]);
const savedShoppingList = await shoppingListStorage.load(listId);
console.log('items after both transactions:', savedShoppingList.items);
```

The code extends the previous use case of overwritten changes. In addition to the file storage component, an instance of the class `ExclusiveAccess` is created. With this utility, the function `addShoppingListItem()` enforces every transaction to wait for earlier ones to finish. As a consequence, concurrently requested item additions can be processed without data loss. However, the resulting execution is almost equivalent to sequential processing, as every transaction waits for other pending ones to finish. While this circumstance is true for the particular implementation approach, controlled concurrency is generally superior. For example, transactions affecting different resources can be executed completely concurrently. Even for a single resource, the ability to enqueue modifications is beneficial. Furthermore, exclusive access is only one of different possible control mechanisms.

Design considerations

There are multiple aspects to consider for the design of an Aggregate. Most importantly, it should align with a specific concept of the respective Domain Model. Typically, an Aggregate encloses a graph of existing Entities and Value Objects, with one of them acting as its root. Introducing a new component that serves as Aggregate Root only makes sense when this part in fact expresses an actual Domain Model concept. Other than these basic recommendations,

there are some more specific and partially technological influencing factors. Most of them are gathered around the concepts of transactions and consistency. The most relevant Aggregate design considerations are explained and illustrated in the following subsections.

Associations

Relationships between separate Aggregates must only be expressed via identifier references. This means that an element of one consistency boundary must never be placed inside another. In terms of persistence, each modification of an Aggregate is made durable within a dedicated transaction. Therefore, a structural consolidation of multiple such boundaries only creates the illusion of transactional behavior. In reality, any alteration affecting more than one transaction is always eventually consistent. For synchronous code without persistence, an object composition of two separate Aggregate boundaries effectively merges them into one. This underlines why the only correct way to express an overarching association is by referencing an identifier. The demand for further knowledge apart from an associated identity either expresses a design issue or the need for synchronization.

True invariants

The existence of invariants significantly influences the design of Aggregates. Generally, invariants fall into one of two categories. There are conceptual rules that are explicitly defined in a Domain Model, and there are technical constraints. For example, requiring non-empty usernames is an important rule, but not necessarily a domain-specific aspect. Regardless of category, when an invariant affects multiple components, they must exist within the same Aggregate. In contrast, elements without shared constraints do not have to. The decisive question is whether the change of one component affects another one somehow. If the answer is yes, both may need to share a common transactional boundary. However, every potential invariant should be questioned for its existence. Computational and informational results are sometimes mistaken for strict transactional rules.

Example: Budget planner

Consider implementing the Domain Model for a budget planner software. Its purpose is to plan expenses within a defined budget. The Domain Model consists of two parts. One is the expense, which encloses a title and units. The second part is the budget plan component, which incorporates a budget and a collection of expenses. For simplicity reasons, the monetary values are expressed as abstract units. The required behavior is to create a plan with a budget and to add individual expenses. On top of that, the software must be able to tell

when the sum of expenses exceeds a budget. There are different interpretation possibilities for this aspect and therefore multiple solution approaches. The key question is whether a budget represents an actual invariant.

The first code provides the Entity type for the expense concept ([run code usage](#)):

Budget planner: Expense Entity

```
class Expense {  
  
    id; title; #units;  
  
    constructor({id, title, units}) {  
        Object.defineProperty(this, 'id', {value: id, writable: false});  
        Object.defineProperty(this, 'title', {value: title, writable: false});  
        this.units = units;  
    }  
  
    set units(units) {  
        if (typeof units !== 'number') throw new Error('invalid units');  
        this.#units = units;  
    }  
  
    get units() { return this.#units; }  
}
```

The class `Expense` captures the Domain Model concept of an individual expense. It defines attributes for an immutable identity, a title and units. The constructor accepts according arguments for the attributes. The implementation as an Entity type enables to make both the title and the units mutable. This approach aims to support use cases such as correcting spelling errors or adjusting units. While the title attribute can be modified directly, the units are adjusted through a setter function. This way, the code can enforce that a given value is an actual number. Another approach would be to implement the concept as Value Object type and perform replacements instead of modifications. In general, there is no strong argument for one or the other implementation approach.

The first implementation of a budget plan encloses complete expense Entities:

Budget planner: Budget plan with Entities

```
class BudgetPlan {  
  
    id; budget; #expenses = [];  
  
    constructor({id, budget}) {  
        if (typeof budget != 'number') throw new Error('invalid budget');  
        Object.defineProperty(this, 'id', {value: id, writable: false});  
        Object.defineProperty(this, 'budget', {value: budget, writable: false});  
    }  
  
    addNewExpense({id, title, units}) {  
        if (this.getLeftUnits() < units) throw new Error('not enough units');  
        this.#expenses.push(new Expense({id, title, units}));  
    }  
  
    changeExpenseTitle(expenseId, newTitle) {  
        const expense = this.#expenses.find(({id}) => id === expenseId);  
        expense.title = newTitle;  
    }  
  
    changeExpenseUnits(expenseId, newUnits) {  
        const expense = this.#expenses.find(({id}) => id === expenseId);  
        const difference = newUnits - expense.units;  
        if (this.getLeftUnits() < difference) throw new Error('not enough units');  
        expense.units = newUnits;  
    }  
  
    getLeftUnits() {  
        const usedUnits = this.#expenses.reduce((sum, {units}) => sum + units, 0);  
        return this.budget - usedUnits;  
    }  
}
```

The class `BudgetPlan` encloses a budget and a list of expenses. Since the component is also an Entity type, its constructor expects a separate identifier. Other than with the expense units, the budget attribute is defined as immutable for the example. The component implementation interprets a budget as an invariant maximum for the sum of contained expenses. Consequently, the command `addNewExpense()` first verifies that creating and adding a new expense does not exceed the limit. The operations `changeExpenseTitle()` and

`changeExpenseUnits()` are facade functions that control the access to contained Entities. The Aggregate design ensures that expense modifications cannot violate the budget invariant. Satisfying the constraint is done with via the function `getLeftUnits()`. This query calculates the difference between the budget and the sum of expenses.

The next code shows an exemplary usage of the budget plan Aggregate ([run code](#)):

Budget planner: Budget plan with Entities usage

```
const plan = new BudgetPlan({id: generateId(), budget: 1000});
const rentExpenseId = generateId();
const foodExpenseId = generateId();

plan.addNewExpense({id: rentExpenseId, title: 'rent', units: 800});
plan.addNewExpense({id: foodExpenseId, title: 'food', units: 250});
plan.changeExpenseUnits(foodExpenseId, 200);

console.log(plan.getLeftUnits());
```

First, a budget plan is created with 1000 as maximum value. Then, two expense identities are defined. Next, two according expenses are added to the Aggregate via the command `addNewExpense()` using the previously defined identifiers. Afterwards, the units of the secondly added expense are decreased. As final step, the left units of the budget plan are logged to the console. When executing the code, an exception is thrown upon the second `addNewExpense()` invocation, as there are not enough units left. Before being able to adjust the second expense in order to fit the budget, the program exits unconditionally. This example raises the question whether the exceeding of a budget is not an invariant, but rather a valid state.

The next approach illustrates an alternative implementation of the budget plan concept:

Budget planner: Budget plan with IDs

```
class BudgetPlan {

    id; budget; #expenses = [];

    constructor({id, budget}) {
        if (typeof budget != 'number') throw new Error('invalid budget');
        Object.defineProperty(this, 'id', {value: id, writable: false});
        Object.defineProperty(this, 'budget', {value: budget, writable: false});
    }
}
```

```
addExpense(expenseId) {
    if (typeof expenseId != 'string') throw new Error('invalid expense id');
    this.#expenses.push(expenseId);
}

getExpenses() { return this.#expenses.slice(); }

}
```

Budget planner: Budget calculator

```
const budgetCalculator = {
    getLeftUnits: (budget, expenses) =>
        budget - expenses.reduce((sum, {units}) => sum + units, 0),
};
```

The updated class `BudgetPlan` works with identifier references instead of Entities. This results in a smaller Aggregate, but also implies a separate one for each expense. Compared to the previous implementation, the constructor function remains identical. The budget is again exposed as public immutable attribute. Query access to the currently contained expense identifiers is provided via `getExpenses()`. The calculation whether expenses exceed a budget is done by the function `getLeftUnits()` of the dedicated service object `budgetCalculator`. This stateless computation is simpler compared to the previous approach. Due to the absence of the invariant protection and the facade functions, the budget plan Entity is less complex. The disadvantage of this implementation is that the logic previously contained in one component is split among multiple parts.

The next code shows an exemplary usage of the budget plan with IDs ([run code](#)):

Budget planner: Budget plan with IDs usage

```
const rentExpenseId = generateId(), foodExpenseId = generateId();
const rentExpense = new Expense({id: rentExpenseId, title: 'rent', units: 800});
const foodExpense = new Expense({id: foodExpenseId, title: 'food', units: 250});

const expenseDatabase = new Map();
expenseDatabase.set(rentExpenseId, rentExpense);
expenseDatabase.set(foodExpenseId, foodExpense);

const plan = new BudgetPlan({id: generateId(), budget: 1000});
plan.addExpense(rentExpenseId);
plan.addExpense(foodExpenseId);
```

```
foodExpense.units = 200;  
  
const expenses = plan.getExpenses().map(id => expenseDatabase.get(id));  
console.log(budgetCalculator.getLeftUnits(plan.budget, expenses));
```

The idea of the second implementation is that an exceeded budget is not an invalid state but an expected condition. Overall, the usage example is more complex than the first one. The expense identifiers in the budget plan must be mapped to Entities before passing them to the `budgetCalculator` service. This is again achieved by maintaining an in-memory database of all existing expenses. The main advantage is that the illustrated use case can execute uninterruptedly, as the query function `getLeftUnits()` never throws an exception. One disadvantage is that the calculator cannot verify the correlation between a budget and expenses. However, this is not necessarily an issue. In fact, it is the responsibility of the architectural layer around the Domain part to orchestrate collaboration between components.

The budget planner example illustrates multiple aspects. For one, domain-specific conditions are not always invariants. Therefore, it is important to challenge every rule that is defined in a Domain Model. Secondly, components that are bound by a common invariant must exist in the same Aggregate. Otherwise, it is impossible to guarantee transactional consistency.

Optimal size

The optimal size for Aggregates depends on various factors. The general recommendation is to make them as small as possible. [Vernon, p. 357] suggests that an Aggregate is ideally “just the Root Entity and a minimal number of attributes and/or Value-typed properties”. While this is valid, the guideline is very generic. Apart from the necessity for Entities with common invariants to share a transactional boundary, there are further relevant aspects. One is the resulting code complexity. Object compositions with multiple levels of elements are more difficult to understand. Another aspect is the performance, especially with regard to persistence. Large data sets are slower to interact with. The third criteria is concurrency. Aggregates with many Entities increase the likelihood of conflicts. In contrast, smaller structures promote transactional success.

Example: Calendar

Consider implementing a simplified calendar software. The Domain Model contains two components. One is the calendar entry, which consists of a title, a description and a date.

All of its attributes must be mutable. Consequently, entries must be distinguishable through identifiers, regardless of their values. The second component is the calendar itself, which is mainly a collection of entries without intrinsic attributes on its own. New items can be added and existing ones can be removed. In addition, there must be a functionality to query specific date ranges. This behavior enables consumers to see all appointments within a certain time frame. There are two different approaches for the design and the implementation of useful Aggregates. Both of them are described in the following paragraphs.

One approach is to form a single Aggregate with the calendar as root and the entries as children. This makes it easy to read data and to query for specific dates. However, it bears issues with regard to the previously explained aspects. Since the Aggregate must act as facade around entries, its code complexity is higher. The performance implications are best illustrated with a calculation: Two years with averagely two appointments per day makes 1460 calendar entries. Even with optimizations, always loading and saving all of them is inefficient. Many use cases only involve single entries anyway. In terms of concurrency, the conceptual flaw is more crucial than the likelihood of conflicts. Modifications to one entry can never affect another one, which makes conflicts impossible.



Sample calculation breakdown

2 appointments x 365 days x 2 years = 1460 entries

1460 entries x 128 bits per UUID = 17920 bits

17920 bits / 8 bits per byte / 1024 bytes per kilobyte = 22.8 kilobytes

An alternative is to implement the calendar and every entry as separate Aggregate. This requires the use of identifier references. The approach solves most of the previous issues. Without a behavioral facade, the code complexity is lower. Performance wise, the data overhead for an entry is only the space required for an identifier. Assuming the use of UUIDs, this makes around 23 kilobytes for the previous example. Concurrent access of different entries is inherently conflict-free. The challenge with this approach is to provide the query functionality. One option is to implement a Domain Service similar to the budget planner example. Another possibility is to utilize Repositories, which are explained in the next chapter. For this example, it is sufficient to mention that there are solutions.

Eventual Consistency

Eventual Consistency describes the circumstance when information is consumed across transactional boundaries and changes are synchronized non-transactionally. This can affect identical representations, derived data structures and even consequential actions. For

example, when charging a credit card, its balance increases immediately. However, in an online account, the information may only be reflected later. Furthermore, the change might trigger a deferred fraud detection mechanism. Eventual Consistency causes distributed data to become temporarily stale after changes. Many times this is acceptable, especially when the synchronization delay is a matter of milliseconds. The consistency model is useful for parts that relate to each other, but cannot exist within the same transactional boundary. The term “eventual” describes that synchronization may take time. However, an update mechanism must never be unreliable.



Explicit consistency models

Even though most software feels like always up-to-date, Eventual Consistency exists in many places. The issue is that some implementations might seem as if they were completely transactional and ignore this aspect. In contrast, the concepts of this book promote to express consistency models explicitly, be it transactional or eventual.

Synchronization strategies

There are multiple options for synchronizing value updates. One is to let dependent components query for changes. As analogy, consider the website of a popular conference. People reload it repeatedly until the ticket sales start. Given enough requests, the website crashes. While the approach has a low complexity for the producer, it performs poorly when dealing with many consumers. Alternatively, the initiator of a modification can actively notify others. For the conference website, all currently connected browsers can be instructed to reload automatically when tickets become available. While this can improve the performance, it requires an according notification mechanism. Independent of the respective approach, the synchronization of data demands meaningful structures to enclose value changes. Domain Events are a fitting candidate for this purpose.

Example: Game server

Consider implementing the Domain Model for a management of game servers. The two main use cases are to check the online status of individual players and to control server capacities. There are two involved components. One is the player part, which encloses an identifier, an online flag and a current server identity. The required behavior is to mark a player as online or as offline. Secondly, there is the game server, which consists of an identifier, a capacity

and a collection of player identities. New ones can be added given there are enough spots, and contained ones can be removed. The described use cases require the online status to exist both in the server and in the player Aggregates. Consequently, the implementation must establish some form of synchronization.



What about data normalization?

An apparent alternative is to store data only once and to compute derived information. Consider the possible approaches and their implications for the example. Keeping the knowledge within the players forces to query them all for determining server capacity. If the information resides in the servers, checking a player status requires a system-wide search. Both approaches do not scale well.

The first code example shows the player Entity type ([run code usage](#)):

Game server: Player Entity

```
class Player {  
  
    id; #isOnline = false; #currentServerId = null;  
  
    constructor({id}) {  
        Object.defineProperty(this, 'id', {value: id, enumerable: true});  
    }  
  
    markAsOnline(serverId) {  
        this.#isOnline = true;  
        this.#currentServerId = serverId;  
    }  
  
    markAsOffline() {  
        this.#isOnline = false;  
        this.#currentServerId = null;  
    }  
  
    get isOnline() { return this.#isOnline; }  
  
    get currentServerId() { return this.#currentServerId; }  
}
```

This is followed by an in-memory database for players ([run code usage](#)):

Game server: Player database

```
const playersById = new Map();

const playerDatabase = {
  save: player => playersById.set(player.id, player),
  load: playerId => playersById.get(playerId),
};
```

The component `Player` expresses the concept of an individual player, which can be marked as online and offline. For every new instance, the private field `#isOnline` is set to `false`, since a player is initially considered offline. The command `markAsOnline()` alters the state by setting the attribute to `true` and saving the passed in server identity. Changing the state back to offline is done via the command `markAsOffline()`. For retrieving the current status, the Entity provides the two accessor functions `isOnline()` and `currentServerId()`. The component `playerDatabase` represents a simple in-memory database to save players and load them via an identifier. Behind the scenes, the Entities are put into a native `Map` object.

The next examples show the definition of two specialized Domain Event types and the game server component ([run code](#)):

Game server: Domain Event types

```
const PlayerAddedToServerEvent = createEventType(
  'PlayerAddedToServer', {serverId: 'string', playerId: 'string'},
);

const PlayerRemovedFromServerEvent = createEventType(
  'PlayerRemovedFromServer', {serverId: 'string', playerId: 'string'},
);
```

Game server: Server Entity

```
class GameServer {

  id; #playerIds = []; #playerCapacity; #eventBus;

  constructor({id, playerCapacity, eventBus}) {
    Object.defineProperty(this, 'id', {value: id, writable: false});
    this.#playerCapacity = playerCapacity;
    this.#eventBus = eventBus;
  }
}
```

```
addPlayer(playerId) {
    if (this.getLeftSpots() <= 0) throw new Error('limit reached');
    this.#playerIds.push(playerId);
    this.#eventBus.publish(
        new PlayerAddedToServerEvent({serverId: this.id, playerId}));
}

removePlayer(playerId) {
    const index = this.#playerIds.indexOf(playerId);
    if (index > -1) this.#playerIds.splice(index, 1);
    this.#eventBus.publish(
        new PlayerRemovedFromServerEvent({serverId: this.id, playerId}));
}

getLeftSpots() { return this.#playerCapacity - this.#playerIds.length; }

}
```

The class `GameServer` is responsible for managing players on a server and ensuring that the capacity is not exceeded. Upon construction, a player identifier collection is initialized with an empty array, and the provided capacity is saved as attribute. The command `addPlayer()` expects an identifier and adds it to the collection, given the server has enough spots left. After the state change, the component creates and publishes the Domain Event `PlayerAddedToServer` with the corresponding identifiers as custom data. The counterpart function `removePlayer()` removes a given value from the player identifier collection. Similar to the other command, it creates and publishes the event `PlayerRemovedFromServer` with the affected identifiers as contained data. Both events represent the means to synchronize each player state and to facilitate Eventual Consistency.

The next code example provides a component for the synchronization of player Entities:

Game server: Player state synchronization

```
const playerStateSynchronization = {
  activate(eventBus) {
    eventBus.subscribe(PlayerAddedToServerEvent.type, event => {
      const player = playerDatabase.load(event.data.playerId);
      player.markAsOnline(event.data.serverId);
    });
    eventBus.subscribe(PlayerRemovedFromServerEvent.type, event => {
      const player = playerDatabase.load(event.data.playerId);
      player.markAsOffline();
    });
  },
};
```

The component `playerStateSynchronization` is responsible for keeping player state eventually consistent with game servers. Its function `activate()` expects an Event Bus as argument and registers two event subscriber operations. One is responsible for processing the event type “`PlayerAddedToServer`”, the other one for the counterpart “`PlayerRemovedFromServer`”. Both callback functions first retrieve the affected player Entity from the database using its identifier from the event data. Afterwards, they execute either the command `markAsOnline()` or `markAsOffline()` to alter the Entity state accordingly. This ensures that whenever the game server functions `addPlayer()` and `removePlayer()` are executed, the respective player Entity state is synchronized.

The last implementation illustrates the combined usage of all previously introduced parts ([run code](#)):

Game server: Example usage

```
const eventBus = new EventBus();

playerStateSynchronization.activate(eventBus);

eventBus.subscribe('PlayerAddedToServer', ({data}) => {
  console.log(`marked player ${data.playerId} as online on ${data.serverId}`);
});

eventBus.subscribe('PlayerRemovedFromServer', ({data}) => {
  console.log(`marked player ${data.playerId} as offline on ${data.serverId}`);
});

const testServer =
```

```
new GameServer({id: generateId(), playerCapacity: 100, eventBus});
const player1 = new Player({id: generateId()});
playerDatabase.save(player1);

testServer.addPlayer(player1.id);
console.log(`added player ${player1.id} to server ${testServer.id}`);
testServer.removePlayer(player1.id);
console.log(`removed player ${player1.id} from server ${testServer.id}`);
```

The first step is the activation of the player state synchronization. This is followed by the registration of two more event subscribers. These functions are used to log game server events. As example Entities, a server and a player are created. The player is saved into the database. Afterwards, it is added to the server and immediately removed again. Both commands are accompanied by a `console.log()` call. Executing the code shows that the addition and the removal messages appear before marking the player as online or offline. The reason is that the Event Bus notifies subscribers asynchronously. This example is an ideal demonstration of Eventual Consistency. Although the player state is temporarily inconsistent, it does not produce errors and is only a matter of milliseconds.

Sample Application: Aggregates

This section illustrates the implementation of Aggregates for the Sample Application. The goal is to establish useful transactional consistency boundaries, primarily in preparation for persistence and concurrency. One preliminary step is to recall the implementation from the previous two chapters. For this purpose, it can be helpful to skim the according sections again. The next step is the identification of Aggregate candidates. This is followed by checking the current implementation for modifications to make. Wherever necessary, a refactoring is applied. For all changes, the relevant code is provided. In general, there is no absolute right or wrong in this activity. Every approach has its own implications. Consequently, the proposed solutions are merely recommendations. Each decision is explained and justified with Domain knowledge and best practices.

Domain Model assessment

There are roughly eleven individual components in the combined Domain Model implementations. All of them are initially provided by Chapter 6. Chapter 7 introduces new shared technical parts, but only refactors existing domain-related code. The search for Aggregate

candidates is best started with a gathering of all Entities and other stateful components. For the user part, there is the user and the e-mail registry. The project context contains Entities for project, team and team member. Looking at the task board part, there are the two components task and task board. All other Domain Model parts are Value Object types, which cannot act as Aggregate Roots. This assessment is the starting point for deciding which Entities form consistency boundaries, and how they relate to each other.



Aggregates across multiple contexts

Although the contexts of the Sample Application are only separated by directories, their layout represents a technological division into subsystems. This leads to the limitation that an Aggregate must not span across these areas. Therefore, it is irrelevant to consider overarching transactional boundaries.

User context

Every user seemingly represents a separate Aggregate with its user Entity as root element. The associated e-mail address and the role are contained immutable values. However, all users modify the same e-mail registry when setting or updating their e-mail address. Assuming that this behavior should be transactional implies two constraints to satisfy. Atomicity demands that either both user and registry are updated or none of them. Secondly, isolation dictates that concurrent e-mail changes do not cause their registry modifications to overwrite each other. Effectively, this creates one large transaction boundary across all users. This design issue can be solved by updating the e-mail registry in a separate transaction after a user modification. In fact, maintaining e-mail address availability is not the responsibility of a user.

The first code shows the definition of a specialized Domain Event type:

User context: Domain Events

```
const UserEmailAddressAssignedEvent = createEventType(
  'UserEmailAddressAssigned', {userId: 'string', emailAddress: 'string'});
```

The second example implements a reworked constructor for the User class:

User context: User Entity constructor with Event Bus

```
constructor(  
    {id, username, emailAddress, password, role, emailAvailability, eventBus}) {  
    verify('valid id', id != null);  
    this.#emailAvailability = emailAvailability;  
    this.#eventBus = eventBus;  
    Object.defineProperty(this, 'id', {value: id, writable: false});  
    Object.assign(this, {username, emailAddress, password, role});  
}
```

The next code shows a refactored e-mail address setter function that publishes a Domain Event ([run code usage](#)):

User context: User Entity e-mail setter

```
set emailAddress(emailAddress) {  
    verify('unused e-mail', this.#emailAvailability.isEmailAvailable(emailAddress));  
    verify('valid e-mail', emailAddress.constructor === EmailAddress);  
    this.#emailAddress = emailAddress;  
    this.#eventBus.publish(new UserEmailAddressAssignedEvent(  
        {userId: this.id, emailAddress: emailAddress.value}));  
}
```

The Domain Event type `UserEmailAddressAssignedEvent` represents an initial assignment or an update of a user's email address. The reworked class `User` ensures a clean transactional separation between individual users and the shared e-mail registry. As additional constructor argument, the Entity type expects an Event Bus. Furthermore, the name for the argument `emailRegistry` is changed to `emailAvailability`. While the injected component stays the same, the term makes more sense in the context of a user. The actual availability check in the operation `set emailAddress()` remains unchanged. In contrast, updating the registry after a state modification is replaced with creating and publishing a "UserEmailAddressAssigned" event instance. This message can be consumed in another component to synchronize the registry state accordingly.

The following code provides a component to consume the previously introduced event and update the email registry:

User context: E-Mail registry synchronization

```
const emailRegistrySynchronization = {

    activate({eventBus}) {
        eventBus.subscribe(UserEmailAddressAssignedEvent.type, (event) => {
            emailRegistry.setUserEmailAddress(event.data.userId,
                new EmailAddress(event.data.emailAddress));
        });
    }

};
```

The next example illustrates the usage of the reworked user context implementation ([run code](#)):

User context: Usage

```
const eventBus = new EventBus();

emailRegistrySynchronization.activate({eventBus});

const emailAddress = new EmailAddress('jd@example.com'), role = new Role('user');

const createNewUser = (username, password) => new User({id: generateId(),
    username, password: createMd5Hash(password), role, emailAddress,
    emailAvailability: emailRegistry, eventBus});

const executeAsync = true;
if (executeAsync) {
    createNewUser('johndoe', 'pw1');
    setTimeout(() => createNewUser('jamesdean', 'pw2'), 0);
} else {
    createNewUser('johndoe', 'pw1');
    createNewUser('jamesdean', 'pw2');
}
```

The object `emailRegistrySynchronization` is responsible for keeping the e-mail registry consistent with user Aggregates. Executing its function `activate()` registers a subscriber for the event “`UserEmailAddressAssigned`” to synchronize the registry state. The exemplary usage creates two users with equal e-mail addresses. Through configuration of the flag `executeAsync` this is done either asynchronously or synchronously. In the asynchronous

case, the implementation behaves as intended. Creating the second user is aborted due to the invariant protection. In contrast, the synchronous execution creates two users with the same e-mail address. Only when processing the second “UserEmailAddressAssigned” event, the registry reports an error for an attempted invariant violation. Overall, the implementation correctly separates transactional boundaries. However, it raises the question how to deal with a potential registry conflict.

Conflict resolution

There are multiple aspects to consider when attempting to provide automated solutions for logical conflicts between eventually consistent components. First, the likelihood of the occurrence must be clarified. For the Sample Application, the only realistic scenario is when a single person accidentally creates multiple users concurrently. Then the question is, whether the conflict causes severe problems. For user instances, it must be ensured that one person cannot access foreign resources. Through the use of conceptual identities, this security issue can be mitigated. The third consideration is about possible action opportunities. One pragmatic approach is to simply report the issue in order for a human to decide on further steps. An automated alternative for the Sample Application would be to deactivate the secondly created user.

The following code shows an exemplary report as reaction to a failed registry update ([run code usage](#)):

User context: E-Mail registry synchronization

```
const emailRegistrySynchronization = {
    activate({eventBus}) {
        eventBus.subscribe(UserEmailAddressAssignedEvent.type, event => {
            try {
                emailRegistry.setUserEmailAddress(
                    event.data.userId, new EmailAddress(event.data.emailAddress));
            } catch (error) {
                console.log('e-mail conflict, report to customer service');
            }
        });
    },
};
```

This solution is sufficient, potentially even for a production environment. Assuming that the Eventual Consistency has a short latency, the actual likelihood of this conflict to occur is negligibly low.

Project context

The original project context implementation already defines well-designed transactional consistency boundaries. Each of the three Entity types team member, team and project is the root of a cleanly separated Aggregate boundary. An individual team member consists of a role and an associated user identity. The team Entity represents a mutable collection of member identifiers. Projects enclose a name and references to a team and a task board. All the relationships between individual Aggregates are expressed with identities. This design is feasible, as the referencing components do not require any further details. Still, one question is whether a larger Aggregate would be more useful than three smaller ones. In the worst case, it would introduce additional complexity and risk concurrency conflicts. Consequently, no refactoring is applied.

Task board context

The task board context forms one large Aggregate for each board together with its contained tasks. Though, for a completely correct implementation, it misses a facade to control task access. One option is to implement this behavior. While this approach requires the least code changes, it increases complexity, performs poorly with persistence, and risks concurrency conflicts. An alternative is to define individual Aggregate boundaries for each Entity type. However, tasks inside a board cannot simply be replaced with identities. The Domain Model implies that a board must be able to tell the status of each working item. This can be solved by replacing tasks with Value Objects that enclose both identifier and status. The replicated state can be synchronized through the facilitation of Eventual Consistency.

The first example provides the Value Object type for describing the combination of a task identifier and a status ([run code usage](#)):

Task board context: Task summary Value Object

```
const TaskSummary = function({taskId, status}) {
  Object.assign(this, {taskId, status});
  Object.freeze(this);
};

TaskSummary.createFromTask = task => {
  verify('valid task', task instanceof Task);
  return new TaskSummary({taskId: task.id, status: task.status});
};
```

The next code shows a reworked task board that works with Value Objects instead of task Entities ([run code usage](#)):

Task board context: Task board with summaries

```
class TaskBoard {  
  
    id; #taskSummaries = [];  
  
    constructor({id}) {  
        verify('valid id', id != null);  
        Object.defineProperty(this, 'id', {value: id, writable: false});  
    }  
  
    addTask(taskSummary) {  
        verify('valid task summary', taskSummary instanceof TaskSummary);  
        verify('task is new', this.#getTaskSummaryIndex(taskSummary.taskId) === -1);  
        this.#taskSummaries.push(taskSummary);  
    }  
  
    removeTask(taskId) {  
        const index = this.#getTaskSummaryIndex(taskId);  
        verify('task is on board', index > -1);  
        this.#taskSummaries.splice(index, 1);  
    }  
  
    getTasks(status = '') {  
        if (status === '') return this.#taskSummaries.slice();  
        return this.#taskSummaries.filter(summary => summary.status === status);  
    }  
  
    updateTaskStatus(taskId, status) {  
        const index = this.#getTaskSummaryIndex(taskId);  
        verify('task is on board', index > -1);  
        this.#taskSummaries.splice(index, 1, new TaskSummary({taskId, status}));  
    }  
  
    containsTask(taskId) { return this.#getTaskSummaryIndex(taskId) > -1; }  
  
    #getTaskSummaryIndex(taskId) {  
        return this.#taskSummaries.findIndex(summary => summary.taskId === taskId);  
    }  
}
```

The class `TaskSummary` is a narrowed-down task representation, consisting of an identifier and a status. This component makes it possible to contain additional details other than an identifier without enclosing a full task Entity. Executing the static factory function `createFromTask()` creates a summary from an existing task instance. The reworked `TaskBoard` implementation replaces the use of Entities with summary Value Objects. Apart from this refactoring, the existing code remains mostly unchanged. Only the command `removeTask()` expects an identifier instead of an object. In addition, three new functions are implemented. Updating the status of a summary due to task changes is done via the operation `updateTaskStatus()`. The query `containsTask()` is required by other components. Lastly, the private helper function `#getTaskSummaryIndex()` is used for internal purposes.

The following code defines a specialized Domain Event type for a task status update:

Task board context: Domain Events

```
const TaskStatusChangedEvent = createEventType(
  'TaskStatusChanged', {taskId: 'string', status: 'string'},
);
```

The next example extend the task Entity type with the publishing of a Domain Event after a status update ([run code usage](#)):

Task board context: Task constructor

```
constructor(
  {id, title, description = '', status = 'todo', assigneeId, eventBus} ) {
  verify('valid id', id != null);
  this.#eventBus = eventBus;
  Object.defineProperty(this, 'id', {value: id, writable: false});
  Object.assign(this, {title, description, status, assigneeId});
}
```

Task board context: Task status setter function with event

```
set status(status) {
    verify('valid status', validStatus.includes(status));
    verify('active task assignee', status !== 'in progress' || !!this.assigneeId);
    this.#status = status;
    this.#eventBus.publish(new TaskStatusChangedEvent({taskId: this.id, status}));
}
```

This is complemented with an extension of the task board synchronization component to synchronize task statuses:

Task board context: Task board synchronization

```
const taskBoardSynchronization = {

    activateTaskAssigneeSynchronization({eventBus, taskBoard}) { /* .. */ },

    activateTaskStatusSynchronization({eventBus, taskBoard}) {
        eventBus.subscribe(TaskStatusChangedEvent.type, event => {
            if (taskBoard.containsTask(event.data.taskId))
                taskBoard.updateTaskStatus(event.data.taskId, event.data.status);
        });
    },
};
```

The class `Task` is changed in multiple ways. For one, its constructor expects an Event Bus as additional argument. Secondly, the function `set status()` creates and publishes the Domain Event “`TaskStatusChanged`” after the state change. The component `taskBoardSynchronization` is extended with the operation `activateTaskStatusSynchronization()` for performing task state synchronization. As arguments, it expects an Event Bus and a task board. The function registers a subscriber callback for the event type “`TaskStatusChanged`” and conditionally updates the affected task summary status. Note that a received Domain Event can refer to any working item and therefore affect any task board. Prior to the execution of the function `updateTaskStatus()`, the operation `containsTask()` is used to check whether a board is affected.

The whole synchronization is best illustrated with a usage example ([run code](#)):

Task board context: Usage with task update

```
const taskBoard = new TaskBoard({id: generateId()});

taskBoardSynchronization.activateTaskStatusSynchronization({eventBus, taskBoard});

const task = new Task({id: generateId(), title: 'write tests', eventBus});
task.assigneeId = generateId();
task.description = 'write unit tests for new feature';
taskBoard.addTask(TaskSummary.createFromTask(task));

task.status = 'in progress';
console.log(taskBoard.getTasks('in progress'));
setTimeout(() => { console.log(taskBoard.getTasks('in progress'))}, 0);
```

The code starts with creating a task board instance. Then, the state synchronization is activated by executing `activateTaskStatusSynchronization()` and passing in an Event Bus and the task board. Next, an exemplary task is instantiated. Afterwards, a summary is generated by executing the factory function `createFromTask()` and passing in the Entity. The returned Value Object is added to the task board. This is followed by updating the status of the task to “in progress”. Finally, the operation `console.log()` is executed twice to illustrate the asynchronicity of the state synchronization. Both calls query the board for tasks that are in progress. The first time, no items are output. Due to the `setTimeout()` call, the second log happens after processing the Domain Event and outputs the expected task summary.

The illustrated implementation is superior to one large Aggregate per task board together with its tasks. This has multiple reasons. The board component has a lower complexity, since there is no need for facade functions around tasks. Also, boards with many working items perform better due to only referencing summaries. Tasks can be used standalone and can be modified concurrently, even when belonging to the same board. However, there are some disadvantages. Eventually consistent task summaries temporally contain outdated information. Furthermore, simultaneous status changes to different tasks can indirectly cause conflicts in one board. This happens when multiple calls of `updateTaskStatus()` are concurrently issued to one Entity. Nevertheless, the solution is more scalable than containing tasks, where every state modification can cause concurrency issues.



Always consider multiple options

As explained earlier, the suggested design is merely a recommendation. Therefore, it is important to not treat it as the single correct solution. Even a larger Aggregate that contains a task board together with its tasks can be feasible. In general, multiple options should be considered instead of exclusively focusing on one.

Eventually consistent context integration

While Aggregates should not span across multiple contexts, an eventually consistent integration is allowed to. In fact, the previous chapter already maintains such a connection between teams and task assignees. The implementation enables that whenever a team member is removed, its assigned tasks are un-assigned. Despite not being explicitly mentioned, this mechanism also facilitates a form of Eventual Consistency. However, in this case, it is not about data being used in multiple places. Rather, as noted earlier in the chapter, the state change of a component causes to trigger a consequential action. The introduction of task summaries requires refactoring of this synchronization, as it cannot directly access task Entities from a board.

The following code example shows a reworked version of the task un-assignment upon team member removal ([run code usage](#)):

Task board context: Task board synchronization

```
const taskBoardSynchronization = {

    activateTaskAssigneeSynchronization({eventBus, taskDatabase}) {
        eventBus.subscribe('TeamMemberRemovedFromTeam', ({data: {teamMemberId}}) => {
            const tasks = taskDatabase.getTasks();
            const assignedTasks = tasks.filter(task => task.assigneeId === teamMemberId);
            assignedTasks.forEach(task => {
                if (task.status === 'in progress') task.status = 'todo';
                task.assigneeId = undefined;
            });
        });
    },
    activateTaskStatusSynchronization({eventBus, taskBoard}) { /* .. */ },
};
```

The final implementation provides an exemplary usage of the task assignee synchronization functionality ([run code](#)):

Task board context: Usage with team member removal

```
const eventBus = new EventBus();

const taskDatabase = new Map();

taskBoardSynchronization.activateTaskAssigneeSynchronization(
  {eventBus, taskDatabase});

const team = new Team({id: generateId(), eventBus});
const teamMemberId = generateId();
team.addMember(teamMemberId);

const task = new Task({id: generateId(), title: 'testing', eventBus});
taskDatabase.set(task.id, task);
task.assigneeId = teamMemberId;
task.description = 'write unit tests for new feature';
task.status = 'in progress';
console.log(`status = ${task.status}, assigneeId = ${task.assigneeId}`);
team.removeMember(teamMemberId);
setTimeout(() => {
  console.log(`status = ${task.status}, assigneeId = ${task.assigneeId}`);
});
```

First, the code defines the `Map` object `taskDatabase`. Next, the task assignee synchronization is activated. Instead of a task board, the reworked version expects a task database for accessing actual Entities. Then, a `Team` Entity is instantiated and a member identifier is added. This is followed by creating a task and saving it into the database. Afterwards, the team member identifier is assigned and the description and status are updated. This is illustrated with an output of the task state. Then, the member identifier is removed from the team. This action causes to create and publish the Domain Event “`TeamMemberRemoved`”, which itself triggers the synchronization. As verification for success, the task state is output again. The usage of `setTimeout()` ensures that the un-assignment is completed.

The illustrated transactional consistency boundaries are an ideal starting point for introducing persistence and concurrency to the Domain Model implementations. The next chapter introduces a concept for persistence mechanisms that focuses on the Domain, instead of favoring technological aspects.

Chapter 9: Repositories

Repositories enable persistence in a meaningful way with regard to the concepts of a Domain Model. Their design goal is to emphasize domain-related aspects as opposed to technological concerns. Through expressive interfaces, the focus lies on components and collections rather than columns and tables, or files and directories. This requires all infrastructural details to be encapsulated inside their implementations. Meaningful Domain behavior is expressed through explicit queries instead of technical search capabilities. Typically, Repositories and Aggregates share a one-to-one relationship, where the root Entities are the directly accessed components. Regardless of the respective alignment, each transactional consistency boundary must have its isolated realm of persistence. Repositories are typically also capable of providing concurrency control, for example through an optimistic locking mechanism.



One-to-one alignment with Aggregates

This chapter exclusively focuses on one Repository per Aggregate type. While there are other valid constellations, this scenario is the most common one. [Vernon, p. 401] also states that there is typically “a one-to-one relationship between an Aggregate type and a Repository”.

Domain Model emphasis

The design goal of Repositories is to put emphasis on the Domain Model instead of technological aspects. Generally, this pattern must not be confused with Data Access Objects (DAO) or Object Relational Mapping (ORM). Despite their common purpose of persistence, the latter concepts can distort the design and the implementation of a Domain Model. Repositories promote a strict and clean separation. Every interface must only reflect existing domain-specific concepts or, alternatively, introduce new ones. Consequently, the used terminology must apply the respective Ubiquitous Language. In contrast, an implementation is allowed to contain technological details, as long as they are encapsulated. Designing a Repository with focus on the Domain Model also avoids extraneous functionalities. Furthermore, providing meaningful querying capabilities prevents Domain concerns from leaking into consumer code.



Use cases for technical interfaces

There are scenarios where it can be useful to provide technical and parametrizable query functionalities. However, the need for this must emerge from the Domain Model. For example, e-mail clients commonly provide the ability to search items based on technical and combinable criteria. In this case, the functionality is specific to the problem space and its Domain.

Design and implementation

The general design recommendation for Repositories is to provide a technology-agnostic interface. On top of that, there are more specific guidelines. [Evans, p. 151] suggests to “provide the illusion of an in-memory collection [...] with more elaborate querying capability”. [Vernon, p. 402] describes two specific design approaches. One is called “collection-oriented” and is equivalent to what Evans describes. The idea is to mimic the appearance and the functionality of collections. Despite possible advantages, the approach requires automatic change tracking, which can cause implicit behavior. The second design is called “persistence-oriented”. With this approach, a Repository may also resemble a collection, but clearly conveys its persistence-specific purpose. While this demands consumers to be aware of persistence, it is more explicit and intention-revealing. The examples in this book exclusively use persistence-oriented Repositories.

Basic functionality

The basic functionality of a Repository is to save and to load individual Aggregate instances through their root Entity. For most components, the save operation is identical and stores a single element. What the querying part consists of, depends on the Domain Model requirements. One essential functionality is to load an object via its identifier. Apart from this, there are endless possibilities. Queries may also respond with computational results such as counts. Repositories should always explicitly convert objects into data and vice versa to differentiate between object representation and persisted information. Generally, it is not advisable to save elements as-is. At a minimum, a conversion ensures data integrity. In JavaScript, Repository interfaces should always be asynchronous, even if the underlying persistence mechanism is synchronous.



What about data deletion?

Apart from saving and loading data, another essential functionality is to delete existing records. This aspect is excluded from the book due to multiple reasons. For one, it causes additional complexity. Secondly, data deletion plays a subordinate role when applying Event Sourcing, which is covered later in the book.

The first code example provides a function to save a file atomically, as explained in the previous chapter ([run code usage](#)):

Filesystem: Write file helper function

```
const writeFileAtomically = async (filePath, content) => {
  const tempPath = `${filePath}-${generateId()}.tmp`;
  await writeFile(tempPath, content);
  await rename(tempPath, filePath);
};
```

The second code shows a simple filesystem-based Repository class:

Repository: Filesystem Repository

```
class FilesystemRepository {

  storageDirectory; #convertToData; #convertToEntity;

  constructor({storageDirectory, convertToData, convertToEntity}) {
    mkdirSync(storageDirectory, {recursive: true});
    Object.defineProperty(
      this, 'storageDirectory', {value: storageDirectory, writable: false});
    this.#convertToData = convertToData;
    this.#convertToEntity = convertToEntity;
  }

  async save(entity) {
    const data = this.#convertToData(entity);
    await writeFileAtomically(this.getFilePath(entity.id), JSON.stringify(data));
  }

  load(id) {
    return readFile(this.getFilePath(id))
      .then(buffer => this.#convertToEntity(JSON.parse(buffer.toString())));
  }
}
```

```
getFilePath(id) {
  if (!id) throw new Error('invalid identifier');
  return `${this.storageDirectory}/${id}.json`;
}

}
```

The class `FilesystemRepository` implements a minimal Repository interface and persists Entities as JSON strings in the filesystem. Its constructor expects three arguments. The parameter `storageDirectory` determines which directory is used as storage location. How to convert an Entity into data and vice versa is controlled with the function parameters `convertToData` and `convertToEntity`. Persisting an object is achieved with the `save()` command. This operation converts an element into data, creates a JSON string and invokes the function `writeFileAtomically()`. The respective `id` attribute is used as filename. Loading an Entity is done via the query `load()`. For this operation, the JSON string is read from a file, de-serialized and transformed into an Entity. Through the configurable converters, the Repository class can be used for arbitrary components.

The next code shows an exemplary usage of the filesystem Repository ([run code](#)):

Repository: Filesystem Repository usage

```
class Counter {

  id; #value;

  constructor({id, start}) {
    Object.defineProperty(this, 'id', {value: id, writable: false});
    this.#value = start;
  }

  increment() { this.#value++; }

  get value() { return this.#value; }

}

const convertToData = counter => ({id: counter.id, value: counter.value});
const convertToEntity = data => new Counter({id: data.id, start: data.value});

const counterRepository = new FilesystemRepository(
  {storageDirectory, convertToData, convertToEntity});
```

```
const counterId = generateId();
const counter = new Counter({id: counterId, start: 42});
counter.increment();
await counterRepository.save(counter);
const savedCounter = await counterRepository.load(counterId);
console.log(`counter ${savedCounter.id} is at ${savedCounter.value}`);
```

The code starts with defining the example Entity class `Counter`. This component represents the concept of a simple numerical counter. As constructor argument, it accepts a start value. The only behavior is to increment by one. For a correctly working Repository, the according converter functions are defined. While `convertToData()` is a one-to-one mapping of information, `convertToEntity()` underlines the need for an explicit transformation. The function correctly instantiates the class `Counter` and uses the field `value` as parameter `start`. Both converters are passed in as arguments to the `FilesystemRepository` constructor. As actual use case, a counter is instantiated, incremented once and persisted. Afterwards, it is loaded again and the retrieved data is output. This illustrates the ability to use the Repository component for any Entity type.

Example: Consultant profile directory

Consider implementing the Domain Model and the persistence mechanism for a directory of consultant profiles. The purpose is to provide recruitment services for companies that need consultants for their projects. The Domain Model consists of two components. One is the consultant profile, which encloses an identifier, a name and a list of skills. Both the identity and the name must be immutable. The skills can be changed by adding new items and removing existing ones. Each entry is represented as simple string value. The second part is the directory of available profiles. This component must provide two querying capabilities. For one, individual instances must be accessible via their identity. Secondly, it must be possible to find all profiles matching a specific skill.

The first code provides the consultant profile Entity type ([run code usage](#)):

Consultant profile directory: Consultant profile Entity

```
class ConsultantProfile {  
  
    id; name; #skills = [];  
  
    constructor({id, name, skills = []}) {  
        Object.defineProperty(this, 'id', {value: id, writable: false});  
        Object.defineProperty(this, 'name', {value: name, writable: false});  
        this.#skills = skills;  
    }  
  
    addSkill(skill) {  
        if (!this.#skills.includes(skill)) this.#skills.push(skill);  
    }  
  
    removeSkill(skill) {  
        const index = this.#skills.indexOf(skill);  
        if (index > -1) this.#skills.splice(index, 1);  
    }  
  
    get skills() { return this.#skills.slice(); }  
  
}
```

The next implementation shows the specialized consultant profile Repository class:

Consultant profile directory: Consultant profile Repository

```
class ConsultantProfileRepository extends FilesystemRepository {  
  
    constructor({storageDirectory}) {  
        super({storageDirectory,  
            convertToData: entity =>  
                ({id: entity.id, name: entity.name, skills: entity.skills}),  
            convertToEntity: data => new ConsultantProfile(  
                {id: data.id, name: data.name, skills: data.skills}));  
    }  
  
    async findAllWithSkill(skill) {  
        const files = await.readdir(this.storageDirectory);  
        const ids = files.map(filename => filename.replace('.json', ''));  
        const entities = await Promise.all(ids.map(id => this.load(id)));  
        return entities.filter(({skills}) => skills.includes(skill));  
    }  
}
```

```
 }  
}
```

The class `ConsultantProfile` expresses the domain-specific concept of a consultant profile as explained previously. Creating an instance requires an identifier and a name. Also, an initial list of skills can be passed in. The commands `addSkill()` and `removeSkill()` enable subsequent state modifications. Executing the getter `get skills()` retrieves all currently contained entries. The component `ConsultantProfileRepository` extends the filesystem Repository. For this purpose, it defines the according converter functions and passes them to the parent constructor. Furthermore, it exposes the query `findAllWithSkill()` to find profiles with a specific skill. This is achieved by determining the identifiers of all available Entities, loading them and filtering the results. Despite possible performance implications of this naive implementation, the operation correctly fulfills the Domain Model requirements.

The last code shows an exemplary usage scenario ([run code](#)):

Consultant profile directory: Usage example

```
const repository = new ConsultantProfileRepository({storageDirectory});  
  
const profile1 = new ConsultantProfile(  
  {id: generateId(), name: 'John Doe', skills: ['development', 'testing']});  
const profile2 = new ConsultantProfile(  
  {id: generateId(), name: 'Jane Doe', skills: ['development', 'design']});  
const profile3 = new ConsultantProfile({id: generateId(), name: 'Jim Doe'});  
  
await Promise.all([repository.save(profile1),  
  repository.save(profile2), repository.save(profile3)]);  
console.log('1st search: ', await repository.findAllWithSkill('development'));  
profile1.removeSkill('testing');  
await repository.save(profile1);  
console.log('2nd search: ', await repository.findAllWithSkill('testing'));
```

First, an instance of the component `ConsultantProfileRepository` is created. Then, three consultant profiles are instantiated, of which two have partially overlapping skills. Each of them is persisted immediately. As first use case, the Repository is queried for Entities with the skill “development”. The resulting objects are output. As preparation for the second scenario, the skill “testing” is removed from the first profile and the change is persisted. Next, the Repository is queried for all entries containing the previously removed skill. This time, the operation yields no result. The usages illustrate that the combination of the Entity and the

Repository interface express the full Domain Model. Every required behavior is implemented accordingly. Also, the usage does not refer to details of the persistence mechanism.

Influences on the Domain Model

The use of Repositories causes additions and changes to a Domain Model and its implementation. For one, Repository interfaces extend the associated abstractions whenever they express new concepts. Still, their implementations must be kept strictly isolated to avoid technological influences. While the behavioral definitions become part of the Domain Model, the underlying specific executions do not. As example, counting unread e-mails is a domain-specific activity expression. Realizing this by querying a database is a technological solution. On top of that, Repositories require that the creation mechanism of storable elements allow an initial state for reconstitution. Typically, this forces Entity constructors to accept arguments for all mutable attributes, even if it deteriorates their code design and expressiveness.



Factories for creation and reconstitution

If the usage of Repositories degrades the expressiveness of a component creation, factories can help to overcome this issue. Instead of exposing a constructor directly for both creating and reconstituting instances, an intention-revealing factory can be implemented. Even though the original component constructor internally still handles both cases, the ambiguity is hidden from the consumer.

Example: Shopping cart

Consider implementing the Domain Model for a simplified shopping cart. The main component consists of an identity and a varying collection of items. Each item is represented as an immutable combination of product identifier and quantity. Normally, every shopping cart starts empty. Consequently, only an identifier is required for the creation. However, for a satisfying shopping experience, it can be useful to provide cart persistence. One key scenario is when a customer starts adding items, but unexpectedly leaves without completing the purchase. Deleting the cart and losing the progress results in a poor user experience. A better approach is to automatically persist its state and reload it when the customer returns. For this purpose, the cart creation must accept a list of initial items.

The first example provides the cart item Value Object type ([run code usage](#)):

Shopping cart: Cart item

```
const CartItem = function({productId, quantity}) {
  Object.freeze(Object.assign(this, {productId, quantity}));
};
```

The next code shows the original cart Entity implementation ([run code](#)):

Shopping cart: Without initial items

```
class ShoppingCart {

  id; #items = [];

  constructor({id}) {
    Object.defineProperty(this, 'id', {value: id, writable: false});
  }

  addItem(item) {
    if (!(item instanceof CartItem)) throw new Error('invalid item');
    this.#items.push(item);
  }

  get items() { return this.#items.slice(); }

}

const initialItems = [
  new CartItem({productId: generateId(), quantity: 3}),
  new CartItem({productId: generateId(), quantity: 4}),
];
const shoppingCart = new ShoppingCart({id: generateId()});
initialItems.forEach(item => shoppingCart.addItem(item));
console.log(shoppingCart.items);
```

The last example provides an altered cart constructor with an argument for initial items ([run code](#)):

Shopping cart: Constructor with initial items

```
constructor({id, initialItems = []}) {  
    Object.defineProperty(this, 'id', {value: id, writable: false});  
    initialItems.forEach(item => this.addItem(item));
```

The class `CartItem` represents an individual item of a shopping cart. Since it is implemented as Value Object, all its attributes are immutable. The class `ShoppingCart` expresses the concept of a shopping cart, enclosing an identity and an item collection. With the initial version, it is impossible to transactionally reconstitute a previously populated cart. Emulating this behavior by creating an empty instance and executing the function `addItem()` repeatedly fails to ensure atomicity. The second constructor implementation eliminates this problem by accepting a list of initial items. Each of them is passed to the operation `addItem()` in order to populate the shopping cart. This approach prevents reconstituted Entities from being temporarily accessible in an incorrect state.

Optimistic Concurrency

There are different strategies for the implementation of Concurrency Control. One is called **Pessimistic Locking**, which is enforced through exclusive resource access, as illustrated in [Chapter 8](#). Another approach is **Optimistic Concurrency**, which exclusively affects the write operations. Reading a resource is always allowed, regardless of concurrency aspects. When attempting to persist changes, the actuality of a modification request is verified first. This causes to only accept an update when it is based on the latest status. An implementation of this approach is the use of version numbers. For every change request, the accompanied version is compared to the currently persisted one. Only if they match, the request is processed and the version is incremented. Otherwise, the request is rejected and an error is reported.



Timestamps as alternative

Instead of version numbers, it is possible to implement Optimistic Concurrency with timestamps. However, there are some challenges to this approach. For one, the timestamp values must have a high precision, potentially even higher than milliseconds. Also, all involved systems must share the exact same machine time. These aspects make the approach generally more complex and error-prone.

When facilitating Optimistic Concurrency, one design question is where to put the concurrency-related information. With regard to transactions, a pragmatic solution is to store

it within the resource records. However, in terms of Domain Modeling and code design there are other aspects to consider. Generally speaking, concurrency-specific information has nothing to do with a Domain Model and should be kept separately. On the other hand, using an attribute directly inside an Entity is a simple solution. It may even be arguable that for some components a version number is a meaningful attribute. In any case, concurrency support should not introduce a lot of additional complexity. In this book, the Repository implementations directly assign versions to Entities, while the components themselves stay agnostic of this attribute.

As preparation for the implementation of Optimistic Concurrency, the following code provides a utility for queuing asynchronous operations ([run code usage](#)):

Repository: Async queue component

```
class AsyncQueue {  
  
    #currentQueuePromise = Promise.resolve();  
  
    enqueueOperation(operation) {  
        this.#currentQueuePromise =  
            this.#currentQueuePromise.then(operation, operation);  
        return this.#currentQueuePromise;  
    }  
  
}
```

The class `AsyncQueue` enqueues operations and executes them one after the other. Adding an operation is done by calling the command `enqueueOperation()`. The provided function should return a `Promise` object, which resolves as soon as its work is completed. The argument is passed as `.then()` handler to the currently latest `Promise` and the result is saved as new latest `Promise`. As the argument is used for both fulfillments and rejections, the queue remains operational in case of an exception. Furthermore, due to the implementation of `Promise.prototype.then()`, it is even possible to enqueue synchronous operations. The return value of the function `enqueueOperation()` is the latest queue `Promise` object. This makes it possible for consumers to wait until their operation is completed and to handle any exceptions.

The next example provides a simple error class for a concurrency conflict ([run code usage](#)):

Repository: Concurrency conflict error

```
class ConcurrencyConflictError extends Error {  
  
    constructor({entityToSave, latestEntity}) {  
        super('ConcurrencyConflict');  
        Object.assign(this, {entityToSave, latestEntity});  
    }  
}
```

The following code implements a concurrency-safe version of the filesystem Repository:

Repository: Concurrency safe filesystem Repository

```
class ConcurrencySafeFilesystemRepository {  
  
    storageDirectory; #convertToData; #convertToEntity; #saveQueueById = new Map();  
  
    constructor({storageDirectory, convertToData, convertToEntity}) {  
        mkdirSync(storageDirectory, {recursive: true});  
        Object.defineProperty(  
            this, 'storageDirectory', {value: storageDirectory, writable: false});  
        this.#convertToData = convertToData;  
        this.#convertToEntity = convertToEntity;  
    }  
  
    save(entity) {  
        if (!this.#saveQueueById.has(entity.id))  
            this.#saveQueueById.set(entity.id, new AsyncQueue());  
        return this.#saveQueueById.get(entity.id).enqueueOperation(async () => {  
            await this.#verifyVersion(entity);  
            const data = this.#convertToData(entity);  
            const dataWithVersion = {...data, version: (entity.baseVersion || 0) + 1};  
            await writeFileAtomically(  
                this.getFilePath(entity.id), JSON.stringify(dataWithVersion));  
        });  
    }  
  
    async load(id) {  
        const data = await readFile(this.getFilePath(id), 'utf-8').then(JSON.parse);  
        return Object.assign(this.#convertToEntity(data), {baseVersion: data.version});  
    }  
}
```

```
getFilePath(id) {
    if (!id) throw new Error('invalid identifier');
    return `${this.storageDirectory}/${id}.json`;
}

async #verifyVersion(entityToSave) {
    const latestEntity = await this.load(entityToSave.id).catch(() => ({}));
    if ((entityToSave.baseVersion || 0) !== (latestEntity.baseVersion || 0))
        throw new ConcurrencyConflictError({entityToSave, latestEntity});
}

}
```

The class `ConcurrencySafeFilesystemRepository` is a filesystem-based Repository component with a version-based Optimistic Concurrency. Saving an Entity has a higher complexity than the previous Repository implementation, as it involves multiple asynchronous and non-transactional steps. Correctly handling this process is done with the help of the `AsyncQueue` component. For every Entity to persist, an individual instance is maintained. Executing the command `save()` adds a new entry to the respective queue. This approach ensures a correct operational ordering and prevents that multiple concurrent save requests interfere with each other. Note that this implementation only works correctly when using a single Repository instance per Entity type. When multiple instances operate concurrently on the same files, it can cause data corruption. The same is true for parallel usage.

The function `save()` initially retrieves the asynchronous queue for the passed in Entity. Then, the actual save process is enqueued, which incorporates multiple steps. First, it compares the base version to the currently persisted value, of which both default to 0. If the versions mismatch, a `ConcurrencyConflictError` is thrown. Otherwise, the passed in Entity is converted to data. Next, an incremented version is assigned. Finally, the result is persisted. The return value is a Promise, which resolves after saving, but rejects for any error. While consumers are able to react to issues, the asynchronous queue ignores them and continues operating. The function `load()` is extended with the responsibility of assigning the `baseVersion` attribute. This avoids requiring the converter function `convertToEntity()` to be aware of versions.



Implicit version attribute

The illustrated approach for the assignment of version numbers creates an implicit attribute on each Domain Model component. This design should be understood as pragmatic solution and is not necessarily an ideal approach. In the rare case that a component itself defines an attribute `baseVersion`, this approach does not work correctly.

The last code shows an exemplary usage of the concurrency-safe repository ([run code](#)):

Repository: Concurrency-safe filesystem Repository usage

```
const convertToData = counter => ({id: counter.id, value: counter.value});
const convertToEntity = data => new Counter({id: data.id, start: data.value});
const counterRepository = new ConcurrencySafeFilesystemRepository(
  {storageDirectory, convertToData, convertToEntity});

const incrementCounter = async id => {
  const counter = await counterRepository.load(id);
  counter.increment();
  return counterRepository.save(counter);
};

const counterId = generateId();
const counter = new Counter({id: counterId, start: 42});
await counterRepository.save(counter);
try {
  await Promise.all([incrementCounter(counterId), incrementCounter(counterId)]);
} catch (error) {
  console.log(error);
}
const savedCounter = await counterRepository.load(counterId);
console.log(`latest persisted value: ${savedCounter.value}`);
```

Similar to the example in the previous section, the code uses the example Entity type `Counter`. Both the converter functions `convertToData()` and `convertToEntity()` remain identical. This emphasizes the ability for the converters to stay version-agnostic. The helper function `incrementCounter()` loads a counter, increments it once and saves the resulting change. For an exemplary usage, a counter Entity is created and persisted. Afterwards, the instance is concurrently modified by executing the operation `incrementCounter()` twice. Any occurring errors are caught and logged to the console. Finally, the counter is loaded again and its current value is output. Executing the code shows that one increment succeeds, while

the other one produces a concurrency conflict. This demonstrates how to use Optimistic Concurrency and also how to make conflicts explicit.

Example: Meetup seat reservation

Consider implementing the Domain Model and the Repositories for a meetup seat reservation software. A meetup is a gathering of a special interest group, where volunteers give talks. While these events are free of charge, they normally have a fixed number of maximum seats. Therefore, the organizers commonly provide a way of making reservations. The Domain Model consists of two different components. One is the reservation, which encloses an e-mail address and a number of seats. The second part is the meetup, which contains an identifier, a title, a capacity and a reservation list. For simplicity reasons, both the title and the capacity are immutable. The required behavior is to add new reservations within the defined capacity and to calculate the number of left seats.

The first example provides a Value Object type for the reservation ([run code usage](#)):

Meetup seat reservation: Reservation Value Object

```
const Reservation = function({creationTime, emailAddress, numberOfSeats}) {
  Object.assign(this, {creationTime, emailAddress, numberOfSeats});
  Object.freeze(this);
};
```

The next code implements the actual meetup Entity component ([run code usage](#)):

Meetup seat reservation: Meetup Entity

```
class Meetup {
  id; title; seatCapacity; #reservations = [];

  constructor({id, title, seatCapacity, initialReservations = []}) {
    Object.defineProperty(this, 'id', {value: id, writable: false});
    Object.defineProperty(this, 'title', {value: title, writable: false});
    Object.defineProperty(
      this, 'seatCapacity', {value: seatCapacity, writable: false});
    initialReservations.forEach(reservation => this.addReservation(reservation));
  }

  addReservation(reservation) {
    if (!(reservation instanceof Reservation)) throw new TypeError('reservation');
```

```
    if (this.getAvailableSeats() < reservation.numberOfSeats)
        throw new Error('not enough seats available');
    this.#reservations.push(reservation);
}

get reservations() { return this.#reservations.slice(); }

getAvailableSeats() {
    const reservedSeats = this.#reservations.reduce(
        (sumOfSeats, reservation) => sumOfSeats + reservation.numberOfSeats, 0);
    return this.seatCapacity - reservedSeats;
}

}
```

The component `Reservation` expresses the reservation concept as Value Object. In addition to its essential attributes, it defines a creation time to avoid an incorrect Equality by Value behavior. For example, consider that a person accidentally reserves one seat instead of two and corrects this with another identical request. With different creation times, the two reservations are distinguishable. The class `Meetup` represents the meetup Entity. All its attributes except the reservation list are defined as immutable. For reconstituting Entities, the constructor accepts initial reservations. Adding a new entry is done by executing the command `addReservation()`. The invariant protection rejects requests for more seats than available. Calculating the remaining count is done with the query `getAvailableSeats()`, which can also be consumed by external components.

As preparation for persistence support, the next example provides custom converter functions for meetup Entities:

Meetup seat reservation: Meetup converters

```
const convertToData = meetup => ({id: meetup.id, title: meetup.title,
    seatCapacity: meetup.seatCapacity,
    reservations: meetup.reservations.map(convertToReservationData)});

const convertToReservationData = ({creationTime, emailAddress, numberOfSeats}) => (
    {creationTime, emailAddress, numberOfSeats});

const convertToEntity = data => new Meetup({id: data.id, title: data.title,
    seatCapacity: data.seatCapacity,
    initialReservations: data.reservations.map(convertToReservation)});
```

```
const convertToReservation = ({creationTime, emailAddress, numberOfSeats}) =>
  new Reservation({creationTime, emailAddress, numberOfSeats});
```

The converter functions encapsulate the required transformation logic for serializing and de-serializing Meetup Entities. They can be used as constructor arguments when instantiating the `ConcurrencySafeFilesystemRepository` component. Converting an Entity to data is done with `convertToData()`, which itself calls the function `convertToReservationData()` for each reservation Value Object. The operation `convertToEntity()` transforms a data object back to a meetup Entity. Translating the reservation data structures into Value Objects is done with the helper function `convertToReservation()`. Although the code is mostly a one-to-one mapping of information, some parts of it contain a use-case-specific conversion. For example, reservations are saved in the field `reservations`, but must be provided as the constructor argument `initialReservations` for reconstitution. Overall, the converter functions ensure to provide the correct structure and types.

The last example in this subsection illustrates an exemplary usage of the meetup seat reservation ([run code](#)):

Meetup seat reservation: Overall usage

```
const meetupRepository = new ConcurrencySafeFilesystemRepository(
  {storageDirectory, convertToData, convertToEntity});

const addReservation = async ({meetupId, emailAddress, numberOfSeats}) => {
  const meetup = await meetupRepository.load(meetupId);
  meetup.addReservation(
    new Reservation({creationTime: Date.now(), emailAddress, numberOfSeats}));
  return meetupRepository.save(meetup);
};

const meetupId = generateId(), title = 'Code & Coffee';
const meetup = new Meetup({id: meetupId, title, seatCapacity: 30});

await meetupRepository.save(meetup);
try {
  await Promise.all([
    addReservation({meetupId, emailAddress: 'jim@example.com', numberOfSeats: 15}),
    addReservation({meetupId, emailAddress: 'joe@example.com', numberOfSeats: 15}),
  ]);
} catch ({entityToSend, latestEntity}) {
  console.log('concurrency conflict');
  console.log('entity to save reservations:', entityToSend.reservations);
```

```
    console.log('latest entity reservations:', latestEntity.reservations);
}
```

The code starts with creating an instance of the `ConcurrencySafeFileSystemRepository` class. Afterwards, it defines the function `addReservation()`, which loads a meetup, adds a reservation and persists the change. As preparation for the use case, a meetup is created and saved. Then, two reservations are added concurrently. One of the operations succeeds, the other one causes a conflict. The error is caught and the reservations from the Entity to save and the last persisted state are output. This shows that both `addReservation()` invocations load the meetup in its initial version, add their reservation and attempt to persist. Similar to the previous generic example, the code demonstrates how to make concurrency conflicts explicit. The next step is to decide what action to take in such a situation.

Retries and conflict resolution

Apart from reporting a concurrency conflict, there are two main different action opportunities. One is to automatically retry an operation until it either succeeds or reaches a number of maximum attempts. This approach has a low complexity and its implementation can be generalized to some extent. However, it promotes implicit behavior and treats conflicts as expected situations. In general, Aggregates and their Repositories should be designed towards transactional success. This means that errors due to concurrent modifications should be the exception. Even more, such occurrences can hint to a design issue in the respective Domain Model. An alternative is to implement a custom conflict resolution. Typically, this involves analyzing the situation and executing an appropriate action. The resulting behavior becomes part of the Domain Model.

Example: Meetup seat reservation enhancement

Consider extending the Domain Model and the implementation for the meetup seat reservation. While the initial version works without conflicts for meetups with moderate demand, the situation is different for popular events. When two or more persons concurrently attempt to add a reservation, only one of them succeeds. Given enough simultaneous requests, this leads to an unsatisfying experience for many users. As mentioned previously, regularly occurring conflicts can hint to an issue in the Domain Model. However, for this example, refactoring the consistency boundaries is not an option. The reservations for a meetup must exist within one Aggregate since they share a common invariant. Their sum must not exceed the maximum seat capacity. One pragmatic solution approach is to implement automatic retries for reservation requests.

The following example implements automatic retries together with an exemplary usage ([run code](#)):

Meetup seat reservation: Automatic retries

```
const meetupRepository = new ConcurrencySafeFilesystemRepository(
  {storageDirectory, convertToData, convertToEntity});

const addReservation = async (data, attempt = 0) => {
  const {meetupId, emailAddress, numberOfSeats} = data;
  if (attempt >= 10) throw new Error('retry limit exceeded');
  const meetup = await meetupRepository.load(meetupId);
  try {
    meetup.addReservation(
      new Reservation({creationTime: Date.now(), emailAddress, numberOfSeats}));
    await meetupRepository.save(meetup);
  } catch (error) {
    if (error.message !== 'ConcurrencyConflict') throw error;
    console.log(`concurrency conflict, executing retry for ${emailAddress}`);
    await addReservation({meetupId, emailAddress, numberOfSeats}, ++attempt);
  }
};

const meetupId = generateId(), title = 'Tea & TypeScript';
const meetup = new Meetup({id: meetupId, title, seatCapacity: 30});

await meetupRepository.save(meetup);
await Promise.all([
  addReservation({meetupId, emailAddress: 'jim@example.com', numberOfSeats: 10}),
  addReservation({meetupId, emailAddress: 'ben@example.com', numberOfSeats: 10}),
  addReservation({meetupId, emailAddress: 'jane@example.com', numberOfSeats: 10}),
]);
const savedMeetup = await meetupRepository.load(meetupId);
console.log(`reservations: ${JSON.stringify(savedMeetup.reservations, null, 2)}`);
```

The usage example is similar to the final implementation from the previous subsection. First, a meetup is created and persisted. Afterwards, there are three `addReservation()` calls, of which none exceeds the seat capacity by itself. Finally, the Entity is loaded again and its reservations are output. The key difference is the implementation of the `addReservation()` command. When adding a reservation fails, the error is caught and analyzed. In case of a concurrency conflict, the function calls itself recursively. This is repeated until the operation succeeds or the maximum retry number is reached. Other errors, such as invariant violation

attempts, are re-thrown. Executing the code shows that one request succeeds immediately, while all others are retried automatically. This behavior improves the user experience for popular meetups.

Although the automatic retry helps to reduce concurrency conflicts, there is another scenario that can still cause user frustration. As example, consider a person attempting to reserve the last five available seats. Simultaneously, another user requests a single seat. The five seat reservation fails due to a concurrency conflict. Even though it is retried, it aborts with a domain-specific error, as there are not enough seats left. The user eventually re-checks the available seat count and sees that there are only four. However, by the time another reservation is issued, there may be other concurrent requests or all seats may be gone. For such a scenario, it can be useful to implement a domain-specific custom conflict resolution.

The next example provides a custom conflict resolution mechanism together with a usage example ([run code](#)):

Meetup seat reservation: Conflict resolution

```
const addReservation = async (data, attempt = 0) => {
  const {meetupId, emailAddress, numberOfWorkers} = data;
  if (attempt >= 10) throw new Error('retry limit');
  const meetup = await meetupRepository.load(meetupId);
  try {
    meetup.addReservation(
      new Reservation({creationTime: Date.now(), emailAddress, numberOfWorkers}));
    await meetupRepository.save(meetup);
  } catch (error) {
    if (error.message !== 'ConcurrencyConflict') throw error;
    const availableSeats = error.latestEntity.getAvailableSeats();
    if (availableSeats === 0) throw new Error('no more workers available');
    const leftSeats = Math.min(numberOfWorkers, availableSeats);
    await addReservation({...data, numberOfWorkers: leftSeats}, ++attempt);
  }
};

const meetupId = generateId(), title = 'DDD deep-dive';
const meetup = new Meetup({id: meetupId, title, seatCapacity: 20});

await meetupRepository.save(meetup);
await addReservation(
  {meetupId, emailAddress: 'jim@example.com', numberOfWorkers: 15});
await Promise.all([
  addReservation({meetupId, emailAddress: 'ben@example.com', numberOfWorkers: 1}),
  addReservation({meetupId, emailAddress: 'john@example.com', numberOfWorkers: 5}),
```

```
]);
const savedMeetup = await meetupRepository.load(meetupId);
console.log(`reservations: ${JSON.stringify(savedMeetup.reservations, null, 2)}`);
```

The reworked implementation of the command `addReservation()` provides a domain-specific conflict resolution mechanism. Every exception other than a concurrency conflict is simply re-thrown. In case of a conflict, the remaining seat count is determined. If there are none, again an exception is thrown. Otherwise, the minimum between the requested and the available seats is calculated and another reservation is issued. Effectively, this is either the original number of seats or all that are left. The exemplary reservation requests resemble the previously described scenario. At the point where the third `addReservation()` call for five seats is executed, there are only four left. The resolution mechanism decides to reserve the remaining seats. This approach exemplifies how to take meaningful action in case of a concurrency conflict.

Interaction with Domain Events

The usage of Domain Events in combination with persistence requires a temporal separation of creation and publishing. For non-transient elements, the single source of truth is their persisted state. This means, something only counts as happened when its associated change is saved. As a consequence, a corresponding event must only be published afterwards. Otherwise, it describes an occurrence that has not happened yet or eventually never does. For example, consider a shopping confirmation e-mail. Every order placement creates a Domain Event that is published regardless of persistence. This triggers a subscriber function to send an e-mail. Consequently, if an order cannot be saved, the confirmation gets sent mistakenly. This illustrates that the distribution of events must only happen after their associated state change was persisted.

The following subsections explain the necessary steps and the different approaches for correctly publishing Domain Events in combination with persistence. This is preceded by the introduction of an example topic. The example is used throughout the subsections to illustrate the individual steps and the different implementation approaches.

Example: Auction platform

Consider working on the implementation of the Domain Model and the Repositories for an auction platform. The relevant functionality is the notification of users when they get

outbid by someone else. The Domain Model implementation consists of two components. One is the bid, which encloses the bidder identity and a price. For brevity reasons, the price is represented as simple numerical value, independent of currency. The second part is the auction itself. This component consists of an identifier, a title, a starting price and a collection of bids. Both the title and the bid collection can change over time. Besides creating an auction, the main required behavior is to place a bid. The operation must provide the means to notify other components about a change.

The first example shows the bid Value Object type ([run code usage](#)):

Auction platform: Bid Value Object

```
const Bid = function({bidderId, price}) {
  Object.freeze(Object.assign(this, {bidderId, price}));
};
```

The next code provides the definition of the Domain Event for when a bid is added:

Auction platform: Domain Event

```
const BidAddedEvent = createEventType('BidAdded',
  {auctionId: 'string', newHighestBidderId: 'string', newHighestPrice: 'number',
   oldHighestBidderId: 'string', oldHighestPrice: 'number'});
```

The third example shows the initial code for the auction Entity type:

Auction platform: Auction Entity

```
class Auction {

  id; title; startingPrice; #bids; #eventBus;

  constructor({id, title, startingPrice, initialBids = [], eventBus}) {
    this.#verifyBids(startingPrice, initialBids);
    this.#eventBus = eventBus;
    this.#bids = initialBids;
    Object.defineProperties(this, {
      id: {value: id, writable: false},
      title: {value: title, writable: true},
      startingPrice: {value: startingPrice, writable: false},
    });
  }
}
```

```
get bids() { return this.#bids.slice(); }

bid(newBid) {
    const highestBid = this.#bids[this.#bids.length - 1];
    this.#verifyBid(newBid, highestBid ? highestBid.price : this.startingPrice);
    this.#bids.push(newBid);
    this.#eventBus.publish(new BidAddedEvent({
        auctionId: this.id,
        newHighestBidderId: newBid.bidderId, newHighestPrice: newBid.price,
        oldHighestBidderId: highestBid ? highestBid.bidderId : '',
        oldHighestPrice: highestBid ? highestBid.price : this.startingPrice,
    }));
}
}

#verifyBids(startingPrice, bids) {
    bids.forEach((bid, index) => this.#verifyBid(bid,
        index > 0 ? bids[index - 1].price : startingPrice));
}

#verfiyBid(bid, priceToOutbid) {
    if (!(bid instanceof Bid)) throw new Error('invalid bid');
    if (bid.price <= priceToOutbid) throw new Error('bid too low');
}
}
```

The class `Bid` expresses the bid concept as Value Object. For describing the occurrence of a bid addition, the specialized Domain Event type `BidAddedEvent` is defined. The auction concept is implemented as the Entity class `Auction`. Its constructor accepts initial bids for reconstituting persisted instances. Also, it expects an Event Bus, which is saved as private attribute. Adding a new bid is done via the command `bid()`. Every submission is first verified by checking its type and validating its price. The first bid is compared to the auction starting price, all other ones to the respective previous bid. In case of success, the function creates and publishes a “`BidAdded`” Domain Event. This message is used to trigger a notification for telling users they were outbid.



Favoring structural consistency

The event type “BidAdded” is used for two separate occurrences, both the initial bid addition and all the following ones. The initial event uses the starting price as the old highest price and leaves out the old bidder identity. This approach favors structural consistency in events, but is less expressive than having two dedicated types.

Collecting events

The creation and the publishing of Domain Events must be separated when working with persistent components. Therefore, all occurring events need to be collected to the point they can be distributed. In its simplest form, this can be implemented with a plain collection. Similar to the topic of concurrency-related information, the question is where to best place this data. When treating it as technical information, it is clear that it should not be contained inside Domain Model components. However, Domain Events are in fact parts of the domain-related abstractions. Consequently, it can be acceptable, yet even useful, to maintain them as a direct attribute. In the following code examples and the sample implementations, all occurring events are kept as a collection inside the Entities.

The first example shows the `bid()` function of a reworked `Auction` class ([run code usage](#)):

Auction platform: Auction Entity with events collection

```
bid(newBid) {
    const highestBid = this.#bids[this.#bids.length - 1];
    this.#verifyBid(newBid, highestBid ? highestBid.price : this.startingPrice);
    this.#bids.push(newBid);
    this.#newDomainEvents.push(new BidAddedEvent({
        auctionId: this.id,
        newHighestBidderId: newBid.bidderId, newHighestPrice: newBid.price,
        oldHighestBidderId: highestBid ? highestBid.bidderId : '',
        oldHighestPrice: highestBid ? highestBid.price : 0,
    }));
}
```

As extension to its existing attributes, the constructor of the `Auction` class defines the private collection `#newDomainEvents`. This transient array is used to gather all newly occurring Domain Events. The reworked command `bid()` creates an instance of the type `BidAddedEvent` and pushes it into the collection instead of publishing it. This way, the component still

expresses that an event occurred, but does not carry the responsibility of triggering its distribution. As a positive side effect, the dependency to the Event Bus is removed from the class. Therefore, the refactoring is not only adding support for persistence, but also reducing the dependencies of the component implementation. For accessing new Domain Events from outside the Entity, the class defines the getter function `get newDomainEvents()`.

Publishing from Repository

One approach for Domain Event distribution in combination with persistence is to let Repositories actively trigger the publishing. This requires their implementation to be aware of events and to know how to retrieve them from the respective Entities. Generally, the exact point in time when events are published is irrelevant, as long as it happens after successfully persisting. As explained in Chapter 7, the distribution ideally happens asynchronously to not rely on the synchronicity of the process. While this approach is a pragmatic solution, it makes Repositories dependent on a publishing mechanism and increases their responsibilities. Architecturally speaking, they should be primarily concerned with persistence, but not necessarily with Domain Event distribution.

The following implementation provides a Repository component that publishes all new Domain Events after saving:

Repository: Event publishing filesystem Repository

```
class EventPublishingFilesystemRepository extends FilesystemRepository {  
  
    #eventBus;  
  
    constructor({storageDirectory, eventBus, convertToData, convertToEntity}) {  
        super({storageDirectory,  
            convertToData: input => this.#omitNewDomainEvents(convertToData(input)),  
            convertToEntity});  
        this.#eventBus = eventBus;  
    }  
  
    #omitNewDomainEvents(input) { return {...input, newDomainEvents: undefined}; }  
  
    async save(entity) {  
        await super.save(entity);  
        for (const event of entity.newDomainEvents) await this.#eventBus.publish(event);  
    }  
}
```

The class `EventPublishingFilesystemRepository` extends the `FilesystemRepository` component and publishes all new events from an Entity after persisting it. As constructor arguments, it expects a storage directory, an Event Bus and converter functions. The directory and the converters are passed along to the parent constructor. Only the function `convertToData()` is decorated with the helper operation `#omitNewDomainEvents()` to avoid persisting Domain Events. The Event Bus is stored as private attribute. Executing the extended command `save()` performs two steps. First, the given Entity is persisted. Afterwards, all its new Domain Events are published in sequence via the Event Bus. The completion of the event publishing is awaited. This enables consumers to know when the overall process is finished. The behavior of the original query `load()` remains unchanged.

The following code implements a helper function to load an auction, add a bid and persist the change:

Auction platform: Add bid helper

```
const addBid = async ({repository, auctionId, bid}) => {
  const auction = await repository.load(auctionId);
  console.log(`adding a bid of ${bid.price}`);
  auction.bid(bid);
  console.log(`saving bid addition of ${bid.price}`);
  await repository.save(auction);
};
```

This is followed by an exemplary usage of the event-publishing Repository together with the auction Entity ([run code](#)):

Auction platform: Usage with event publishing Repository

```
const repository = new EventPublishingFilesystemRepository(
  {storageDirectory, eventBus, convertToData, convertToEntity});

eventBus.subscribe(BidAddedEvent.type, event =>
  console.log(`got BidAdded event with ${event.data.newHighestPrice}`));

const auctionId = generateId(), bidderId = generateId();
const auction = new Auction({id: auctionId, title: 'Laptop', startingPrice: 100});

await repository.save(auction);
await addBid({repository, auctionId, bid: new Bid({bidderId, price: 110})});
await addBid({repository, auctionId, bid: new Bid({bidderId, price: 120})});
const savedAuction = await repository.load(auctionId);
console.log('final loaded bids:', savedAuction.bids);
```

First, the code creates an instance of the `EventPublishingFilesystemRepository` class and passes in according converter functions as constructor arguments. Next, a subscriber function for the “BidAdded” event is registered to demonstrate the notification possibility for outbid users. The actual usage starts with defining identifiers for the auction and the bidder, as well as creating an auction Entity. After saving the initial state, the command `addBid()` is executed twice for simulating bid additions. The helper function loads the auction, adds a bid and persists the change. Also, it logs status messages for informational purposes. Finally, the auction is loaded again and its current bids are output. Executing the code shows that the events are not published directly after each state modification, but only after saving.

Guaranteed delivery through persistent events

The publishing mechanism of Domain Events must be reliable and ensure guaranteed delivery. Typically, the combination of saving a change and publishing an event is not transactional. This means, independent of the order, the latter operation can fail without affecting the first one. As example, reconsider the shopping confirmation e-mail. Publishing an event after successfully persisting avoids sending out notifications for orders that cannot be saved. However, if the event distribution fails, no notification is sent, even though an order was persisted. As explained in Chapter 7, this can be solved with persistent queues. One simple approach for the publisher part is the **Outbox** pattern. With every Entity change, Domain Events are transactionally saved inside the same data storage and their publishing is handled separately.

As preparation for a guaranteed delivery, the following code provides the implementation of an event-saving Repository ([run code usage](#)):

Repository: Event-saving filesystem Repository

```
class EventSavingFilesystemRepository extends FilesystemRepository {  
  
    #convertToData;  
  
    constructor({storageDirectory, convertToData, convertToEntity}) {  
        super({storageDirectory, convertToData, convertToEntity});  
        this.#convertToData = convertToData;  
    }  
  
    async save(entity) {  
        const data = {...this.#convertToData(entity), newDomainEvents: undefined};  
        const savedEvents = await this.loadDomainEvents(entity.id);  
        const allEvents = savedEvents.concat(entity.newDomainEvents);  
    }  
}
```

```
    const fullData = {...data, domainEvents: allEvents};
    const filePath = this.getFilePath(entity.id);
    await writeFileAtomically(filePath, JSON.stringify(fullData));
}

loadDomainEvents(id) {
    return readFile(this.getFilePath(id), 'utf-8')
        .then(JSON.parse).then(data => data.domainEvents || []).catch((() => []));
}

}
```

The class `EventSavingFilesystemRepository` extends the `FilesystemRepository` component and enables persistence for Entities with their Domain Events. Its constructor forwards all arguments to the parent class. Also, it defines a separate private field for the converter operation `convertToData()`. The `save()` command completely replaces the implementation of the parent class. As the original, it converts an Entity to data and omits the attribute `newDomainEvents`. In addition, it loads all persisted events, appends newly occurred ones and attaches the result to the data. Finally, it saves the information to the filesystem. The `load()` operation again remains unchanged. For retrieving all saved events of an Entity, the class provides the operation `loadDomainEvents()`, which is also re-used internally. Note that loading an Entity does not include its persisted Domain Events.

The next example implements a Repository component that enables fail-safe event publishing:

Repository: Fail-safe event publishing filesystem Repository

```
class FailSafeEventPublishingFilesystemRepository
    extends EventSavingFilesystemRepository {

    #publishedEventIdsDirectory; #eventBus;

    constructor({storageDirectory, eventBus, convertToData, convertToEntity}) {
        super({storageDirectory, convertToData, convertToEntity});
        this.#publishedEventIdsDirectory = `${storageDirectory}/last-published-events`;
        this.#eventBus = eventBus;
        mkdirSync(this.#publishedEventIdsDirectory, {recursive: true});
    }

    async save(entity) {
        await super.save(entity);
```

```
    await Promise.all(entity.newDomainEvents.map(
      domainEvent => this.#eventBus.publish(domainEvent)));
    await this.#saveLastPublishedEventId(entity.id, entity.newDomainEvents);
  }

  async publishOverdueDomainEvents(entityId) {
    const allEvents = await super.loadDomainEvents(entityId);
    const lastPublishedEventId = await this.#getLastPublishedEventId(entityId);
    const eventsToPublish = allEvents.slice(
      allEvents.findIndex(event => event.id === lastPublishedEventId) + 1);
    for (const event of eventsToPublish) await this.#eventBus.publish(event);
    await this.#saveLastPublishedEventId(entityId, eventsToPublish);
  }

  #getLastPublishedEventId(entityId) {
    const filePath = `${this.#publishedEventIdsDirectory}/${entityId}`;
    return readFile(filePath, 'utf-8').catch(() => '');
  }

  async #saveLastPublishedEventId(entityId, publishedEvents) {
    if (publishedEvents.length === 0) return;
    const filePath = `${this.#publishedEventIdsDirectory}/${entityId}`;
    await writeFile(filePath, publishedEvents[publishedEvents.length - 1].id);
  }
}
```

The class `FailSafeEventPublishingFilesystemRepository` extends the previous event-saving Repository component with guaranteed event publishing. Its constructor again forwards all given arguments to the parent class. Also, it creates a subdirectory for the identifier of each respectively last published event per Entity. The `save()` command extends the original version by publishing new events and executing the function `#saveLastPublishedEventId()`. This operation saves the identity of the last published event in a file, using the Entity identifier as filename. Recovering from a distribution failure is achieved with the command `publishOverdueDomainEvents()`. The operation loads all persisted events of an Entity and issues the unpublished ones for distribution. While the implementation guarantees event publishing, it can lead to duplicates in rare cases. Effectively, the class establishes an “at least once” publishing guarantee.



Outbox collection

The illustrated Repository implementation persists each Entity and its Domain Events in the same file to achieve a transactional behavior. When using a database, one common approach is to create a separate outbox collection for events to publish. Every state change and the saving of associated events is combined into one transaction. After publishing, the respective events are marked accordingly.

As preparation for the usage example, the following code provides a function to simulate an Event Bus publishing failure:

Auction platform: Make Event Bus publish fail helper

```
const makeEventBusPublishFailOnce = eventBus => {
  const originalPublish = eventBus.publish;
  eventBus.publish = () => {
    eventBus.publish = originalPublish;
    throw new Error('publish failed');
  };
};
```

The next implementation shows an exemplary usage of the fail-safe event publishing Repository together with the auction Entity ([run code](#)):

Auction platform: Usage with fail safe event publishing Repository

```
const repository = new FailSafeEventPublishingFilesystemRepository(
  {storageDirectory, eventBus, convertToData, convertToEntity});

eventBus.subscribe(BidAddedEvent.type, event =>
  console.log(`got BidAdded event with ${event.data.newHighestPrice}`));

const auctionId = generateId(), bidderId = generateId();
const auction = new Auction({id: auctionId, title: 'Laptop', startingPrice: 100});

await repository.save(auction);
await addBid({repository, auctionId, bid: new Bid({bidderId, price: 110})});
await makeEventBusPublishFailOnce(eventBus);
try {
  await addBid({repository, auctionId, bid: new Bid({bidderId, price: 120})});
} catch (error) {
  console.log(error.message);
}
```

```
const savedAuction = await repository.load(auctionId);
console.log('current bids:', savedAuction.bids);
console.log('publishing overdue events');
repository.publishOverdueDomainEvents(auctionId);
```

The code starts with instantiating the class `FailSafeEventPublishingFilesystemRepository`. Next, a subscriber for the “`BidAdded`” event is registered. This is followed by defining all necessary identifiers and creating an auction Entity. After initially persisting, the previously introduced function `addBid()` is executed twice, with one call to `makeEventBusPublishFailOnce()` in between. Every occurring error is logged through a catch handler. As next step, the Entity is re-loaded and its current bids are output. The example ends with calling the function `publishOverdueDomainEvents()`. Running the code shows that the second `addBid()` call successfully saves the changes, but never distributes the associated event. Only when publishing the overdue Domain Events, the subscriber function is executed. This demonstrates the guaranteed event publishing, even in case of software failures.

Separation of persistence and publishing

Instead of triggering the publishing, Repositories can expose access to persisted Domain Events and let other components handle the distribution. After all, Repositories should be concerned with saving and loading data, ensuring its integrity and eventually supporting concurrency. However, event publishing should not be their responsibility. Exposing read access to persisted Domain Events fits within their scope. On top of that, they can provide Entity change notifications, which help to avoid active polling. Another component can watch for changes, access persisted events, determine new ones and notify interested parties. This can be done by the Event Bus or a separate part that triggers the publishing. With this design, consumers must manage their own publishing state. The advantage is that Repositories stay agnostic of the distribution.

As preparation for a separate publishing component, the event-saving filesystem Repository is extended with the following function: ([run code usage](#)):

Repository: Event saving filesystem Repository extension

```
subscribeToEntityChanges(subscriber) {
    fs.watch(this.storageDirectory, (_, filename) => {
        if (filename.endsWith('.json')) subscriber(filename.replace('.json', ''));
    });
}
```

The next example provides a component to consume Domain Events and trigger their publishing via the Event Bus:

Domain Event: Domain Event Publisher

```
class DomainEventPublisher {

    #repository; #eventBus; #publishingQueue; #publishedEventIdsDirectory;

    constructor({repository, eventBus, publishedEventIdsDirectory}) {
        this.#repository = repository;
        this.#eventBus = eventBus;
        this.#publishingQueue = new AsyncQueue();
        this.#publishedEventIdsDirectory = publishedEventIdsDirectory;
        mkdirSync(this.#publishedEventIdsDirectory, {recursive: true});
    }

    activate() {
        this.#repository.subscribeToEntityChanges(
            id => this.publishDueDomainEvents(id));
    }

    publishDueDomainEvents(entityId) {
        this.#publishingQueue.enqueueOperation(async () => {
            const allEvents = await this.#repository.loadDomainEvents(entityId);
            const lastPublishedEventId = await this.#getLastPublishedEventId(entityId);
            const eventsToPublish = allEvents.slice(
                allEvents.findIndex(event => event.id === lastPublishedEventId) + 1);
            await Promise.all(eventsToPublish.map(event => this.#eventBus.publish(event)));
            await this.#saveLastPublishedEventId(entityId, eventsToPublish);
        });
    }

    _getLastPublishedEventId(entityId) { /* .. */ }

    _saveLastPublishedEventId(entityId, publishedEvents) { /* .. */ }
}
```

}

The event-saving Repository is extended with the command `subscribeToEntityChanges()` to notify subscribers of Entity modifications. Its implementation uses the `fs.watch()` functionality to detect filesystem changes. Each notification includes the affected Entity identifier as argument. The component `DomainEventPublisher` watches for new Domain Events and triggers their publishing. Its constructor expects a Repository, an Event Bus and a publishing state directory. Executing the operation `activate()` registers an Entity change subscriber that issues the event publishing. The function `publishDueDomainEvents()` loads all events of an Entity, publishes due items and updates their status. For concurrency support, the process is enqueued in an `AsyncQueue` instance. This command can also be used for failure recovery. Both the operations `#getLastPublishedEventId()` and `#saveLastPublishedEventId()` are identical to the implementation of the `FailSafeEventPublishingFilesystemRepository` class.

The final example shows a usage of the extended Repository implementation and the publisher component for the auction platform ([run code usage](#)):

Auction platform: Usage with event saving publishing Repository and publisher

```
const repository = new EventSavingFilesystemRepository(  
  {storageDirectory, convertToData, convertToEntity});  
  
const domainEventPublisher = new DomainEventPublisher({  
  publishedEventIdsDirectory: `${storageDirectory}/published-event-ids`,  
  eventBus, repository,  
});  
  
domainEventPublisher.activate();  
  
eventBus.subscribe(BidAddedEvent.type, event =>  
  console.log(`got BidAdded event with ${event.data.newHighestPrice}`));  
  
const auctionId = generateId(), bidderId = generateId();  
const auction = new Auction({id: auctionId, title: 'Laptop', startingPrice: 100});  
  
await repository.save(auction);  
await addBid({repository, auctionId, bid: new Bid({bidderId, price: 110})});  
await addBid({repository, auctionId, bid: new Bid({bidderId, price: 120})});  
console.log('adding bids completed');
```

Again, the code starts with creating an instance of the event-saving filesystem Repository

component. Afterwards, the class `DomainEventPublisher` is instantiated and enabled by executing `activate()`. Also, a subscriber function for the “`BidAdded`” event is registered. Then, all necessary identifiers are defined and an example auction is created and persisted. The actual use case consists of two subsequent invocations of the `addBid()` helper. Running the code can produce different results, depending on the execution environment. This is due to using `fs.watch()` for Entity change notifications. The event publishing may only happen after both bid additions are completed. In this case, Node.js merges the filesystem changes into one notification. Overall, this example demonstrates how to combine persistence and reliable Domain Event publishing without polluting Repository implementations.

Sample Application: Repositories and event publishing

This sample section illustrates the implementation of concurrency-safe Repositories and Domain Event distribution with guaranteed publishing for the Sample Application. The result is a fully functional Domain Model implementation together with persistence support. Similar to the other chapters, this section is a stepping stone towards a software using CQRS and Event Sourcing. For brevity reasons, the provided Repositories only contain exemplary functionality and are not feature complete. Their implementations re-use numerous functionalities introduced earlier in the chapter. As a consequence, this section can be difficult to understand without reading the preceding content. The Sample Application contexts are discussed in the order of their additional code complexity, starting with the lowest. Each context-specific Repository is placed in a directory called “`persistence`” within the respective “`infrastructure`” directory.



Repositories with inheritance

The Repositories in this subsection are implemented as subclasses of a base component. Note that this is one possible approach out of many and does not represent an implementation recommendation. Independent of specific styles, the goal for a Repository component is to encapsulate the persistence mechanism of an Entity type.

Shared infrastructure code

Similar to the other chapters, this section introduces additional shared infrastructural functionalities to the Sample Application. The previously shown concurrency-safe Repos-

itory and event-saving Repository are combined into one larger component. The resulting class carries further the name `ConcurrencySafeFilesystemRepository`. Naturally, its implementation re-uses the filesystem helper function `writeFileAtomically()`, the `AsyncQueue` component and the `ConcurrencyConflictError` type. Also, it implements the custom query `loadAll()` to load all persisted Entities within one operation. Note that the full Repository code is not shown separately, as it is mainly a combination of already known components. For guaranteed Domain Event publishing, the class `DomainEventPublisher` is integrated into the Sample Application codebase.

The first code shows the `loadAll()` function of the combined concurrency-safe filesystem Repository class ([run code usage](#)):

Infrastructure code: Additional Repository function

```
async loadAll() {  
    const files = await.readdir(this.storageDirectory);  
    const jsonFiles = files.filter(filename => filename.endsWith('.json'));  
    const ids = jsonFiles.map(filename => filename.replace('.json', ''));  
    return Promise.all(ids.map(id => this.load(id)));  
}
```

As helper for subsequent usage examples, the next code provides a function that resolves after a given amount of time ([run code usage](#)):

Infrastructure code: Timeout function

```
const timeout =  
duration => new Promise(resolve => setTimeout(resolve, duration));  
  
console.log('start');  
await timeout(1000);  
console.log('continue');
```

Project context

The project context implementation requires the least complex source code modifications and additions to be made for enabling persistence. Both the Entity classes `TeamMember` and `Project` and the Value Object type `Role` remain unchanged. In contrast, the class `Team` is extended with multiple aspects. For one, it defines the collection `#newDomainEvents` and the getter `newDomainEvents()` for event persistence and distribution. The constructor

argument for the Event Bus is removed. Furthermore, the component accepts initial member identifiers for Entity reconstitution. When removing a team member, the Domain Event “TeamMemberRemoved” is created and added to the event collection. For each of the Entity types of the project context, a Repository is implemented as subclass of the `ConcurrencySafeFilesystemRepository` component.

The first examples shows the refactored parts of the `Team` Entity:

Project context: Team constructor

```
constructor({id, initialTeamMemberIds = []}) {
    verify('valid id', id != null);
    Object.defineProperty(this, 'id', {value: id, writable: false});
    this.#teamMemberIds = [];
    initialTeamMemberIds.forEach(teamMemberId => this.addMember(teamMemberId));
}
```

Project context: Team member removal

```
removeMember(teamMemberIdToRemove) {
    const indexToRemove = this.#teamMemberIds.indexOf(teamMemberIdToRemove);
    if (indexToRemove === -1) return;
    this.#teamMemberIds.splice(indexToRemove, 1);
    this.#newDomainEvents.push(new TeamMemberRemovedFromTeamEvent(
        {teamId: this.id, teamMemberId: teamMemberIdToRemove}));
}
```

The next examples provide all Repository classes for the project context:

Project context: Team member Repository

```
class TeamMemberRepository extends ConcurrencySafeFilesystemRepository {

    constructor({storageDirectory}) {
        super({storageDirectory,
            convertToData: entity => ({id: entity.id, userId: entity.userId,
                role: entity.role.name}),
            convertToEntity: data => new TeamMember(
                {id: data.id, userId: data.userId, role: new Role(data.role)}));
    }

    async findTeamMembersByUser(userId) {
        const members = await this.loadAll();
```

```
    return members.filter(
      member => member.userId === userId).map(member => member.id);
  }

}
```

Project context: Team Repository

```
class TeamRepository extends ConcurrencySafeFilesystemRepository {

  constructor({storageDirectory}) {
    super({storageDirectory,
      convertToData: ({id, teamMemberIds}) => ({id, teamMemberIds}),
      convertToEntity: data => new Team(
        {id: data.id, initialTeamMemberIds: data.teamMemberIds}));
  }

  async findTeamByTeamMember(teamMemberId) {
    const teams = await this.loadAll();
    return teams.find(team => team.teamMemberIds.includes(teamMemberId));
  }

}
```

Project context: Project Repository

```
class ProjectRepository extends ConcurrencySafeFilesystemRepository {

  constructor({storageDirectory}) {
    super({storageDirectory,
      convertToData: ({id, name, ownerId, teamId, taskBoardId}) =>
        ({id, name, ownerId, teamId, taskBoardId}),
      convertToEntity: ({id, name, ownerId, teamId, taskBoardId}) =>
        new Project({id, name, ownerId, teamId, taskBoardId}));
  }

  async findProjectsByOwner(ownerId) {
    const projects = await this.loadAll();
    return projects.find(project => project.ownerId === ownerId);
  }

  async findProjectByTeam(teamId) {
```

```
    const projects = await this.loadAll();
    return projects.find(project => project.teamId === teamId);
}

}
```

The classes `TeamMemberRepository`, `TeamRepository` and `ProjectRepository` extend the generic `Repository` component, define custom converter functions and implement additional custom queries. This is to support the use case of finding all projects that a user is a team member of. The operation `findTeamMembersByUser()` on the `TeamMemberRepository` class returns all team member representations for a user identity. Finding a team for a given member identifier is done with the query `findTeamByTeamMember()` on the `TeamRepository` component. The operation `findProjectByTeam()` on the `ProjectRepository` class yields the project for a given team. On top of that, the query `findTeamsByOwner()` on the `ProjectRepository` type returns all owned projects for an individual user. Independent of their technology-specific implementations, the query signatures purely express domain-specific behavior.

The next code shows an exemplary usage of the refactored context together with the `DomainEventPublisher` component for publishing Domain Events ([run code usage](#)):

Project context: Overall usage

```
const teamMemberRepository = new TeamMemberRepository(
  {storageDirectory: `${rootStorageDirectory}/team-member`});
const teamRepository = new TeamRepository(
  {storageDirectory: `${rootStorageDirectory}/team`});
const projectRepository = new ProjectRepository(
  {storageDirectory: `${rootStorageDirectory}/project`});

const publisher = new DomainEventPublisher({
  repository: teamRepository, eventBus,
  publishedEventIdsDirectory: `${rootStorageDirectory}/team/published-event-ids`,
});

publisher.activate();

eventBus.subscribe(TeamMemberRemovedFromTeamEvent.type, console.log);

const findProjectsByCollaboratingUser = async userId => {
  const teamMemberIds = await teamMemberRepository.findTeamMembersByUser(userId);
  const teams = await Promise.all(
    teamMemberIds.map(id => teamRepository.findTeamByTeamMember(id)));
}
```

```
return await Promise.all(
  teams.map(({id}) => projectRepository.findProjectByTeam(id)));
};

const [projectId, teamId, userId, teamMemberId, taskBoardId] =
  [generateId(), generateId(), generateId(), generateId(), generateId()];
const role = new Role('developer');
const team = new Team({id: teamId});
const project = new Project(
  {id: projectId, ownerId: userId, name: 'test', teamId, taskBoardId});
const teamMember = new TeamMember({id: teamMemberId, userId, role});

await Promise.all([teamRepository.save(team), projectRepository.save(project),
  teamMemberRepository.save(teamMember)]);

const emptyTeam = await teamRepository.load(teamId);
emptyTeam.addMember(teamMemberId);
await teamRepository.save(emptyTeam);
console.log(await projectRepository.findProjectsByOwner(userId));
console.log(await findProjectsByCollaboratingUser(userId));
const populatedTeam = await teamRepository.load(teamId);
populatedTeam.removeMember(teamMemberId);
teamRepository.save(populatedTeam);
```

User context

Although the user context implementation contains only one Aggregate, it requires more diverse adaptations compared to the previous context. For design reasons, the e-mail registry object is replaced with a class. The Aggregate Root Entity class `User` is changed in multiple ways. Its constructor also removes the Event Bus argument and introduces an event collection. Furthermore, it accepts the parameter `isExistingUser` to differentiate between new and existing Entities. For each new user, the e-mail setter operation is executed, which causes to create and add a “`UserEmailAddressAssigned`” event. For existing users, the attribute value is set directly. Both the types `EmailAddress` and `Role` remain unchanged. The factory `UserFactory` is introduced for encapsulating creation and reconstitution. For the persistence, the component `UserRepository` is implemented as subclass of `ConcurrencySafeFilesystemRepository`.

The first examples show the refactored constructor and the e-mail setter function of the `User` Entity:

User context: User constructor

```
constructor({id, username, emailAddress, password, role,
  emailAvailability, isExistingUser}) {
  verify('valid id', id != null);
  this.#emailAvailability = emailAvailability;
  Object.defineProperty(this, 'id', {value: id, writable: false});
  Object.assign(this, {username, password, role});
  if (isExistingUser) this.#emailAddress = emailAddress;
  else this.emailAddress = emailAddress;
}
```

User context: User e-mail address setter function

```
set emailAddress(emailAddress) {
  verify('unused e-mail', this.#emailAvailability.isEmailAvailable(emailAddress));
  verify('valid e-mail', emailAddress.constructor === EmailAddress);
  this.#emailAddress = emailAddress;
  this.#newDomainEvents.push(new UserEmailAddressAssignedEvent(
    {userId: this.id, emailAddress: emailAddress.value}));
}
```

The next examples provide the class `UserFactory` and the Repository component `UserRepository`:

User context: User factory

```
class UserFactory {

  #emailRegistry;

  constructor({emailRegistry}) {
    this.#emailRegistry = emailRegistry;
  }

  createUser({id, username, emailAddress, password, role}) {
    return new User({id, username, emailAddress, password, role,
      emailAvailability: this.#emailRegistry, isExistingUser: false});
  }

  reconstituteUser({id, username, emailAddress, password, role}) {
    return new User({id, username, emailAddress, password, role,
      emailAvailability: this.#emailRegistry, isExistingUser: true});
  }
}
```

```
}
```

User context: User repository

```
class UserRepository extends ConcurrencySafeFilesystemRepository {

    constructor({storageDirectory, userFactory}) {
        super({
            storageDirectory,
            convertToData: ({id, username, emailAddress, password, role: {name: role}}) =>
                ({id, username, emailAddress: emailAddress.value, password, role: role}),
            convertToEntity: ({id, username, emailAddress, password, role}) =>
                userFactory.reconstituteUser({id, username, password,
                    emailAddress: new EmailAddress(emailAddress), role: new Role(role)}),
        });
    }

    async findUserByEmail(emailAddress) {
        const users = await this.loadAll();
        return users.find(user => user.emailAddress.equals(emailAddress));
    }
}
```

The class `UserFactory` defines functions for instantiating two different types of `User` objects. Creating a new and previously unsaved Entity is done with the operation `createNewUser()`. Internally, it invokes the user constructor with the parameter `isNewUser` set to true. This ensures that the Domain Event “`UserEmailAddressAssigned`” is created and added to the event collection. In contrast, the function `reconstituteUser()` is responsible for re-creating an existing Entity. In this case, the parameter `isNewUser` is set to false, which causes no new Domain Event to be added. The component `UserRepository` extends the class `ConcurrencySafeFilesystemRepository` and passes custom converters to the parent constructor. Also, it defines the specialized query `findUserByEmail()`. This enables the identification of an Entity via its assigned e-mail address, for example when authenticating a user.

The following code shows an exemplary usage of the User context with regard to persistence and event publishing ([run code usage](#)):

User context: Overall usage

```
const emailRegistry = new EmailRegistry();
const userFactory = new UserFactory({emailRegistry});
const userRepository = new UserRepository({storageDirectory, userFactory});
const publisher = new DomainEventPublisher({repository: userRepository, eventBus,
    publishedEventIdsDirectory: `.${storageDirectory}/published-event-ids`});

publisher.activate();

eventBus.subscribe(UserEmailAddressAssignedEvent.type, console.log);

const setupEmailRegistry = emailRegistry => {
    emailRegistrySynchronization.activate({eventBus, emailRegistry});
    return userRepository.loadAll().then(users => users.forEach(
        user => emailRegistry.setUserEmailAddress(user.id, user.emailAddress)));
};

const id = generateId(), randomSuffix = `${Date.now()}`;
const username = 'johndoe', role = new Role('user'), password = 'pw1';
const emailAddress = new EmailAddress(`jd${randomSuffix}@example.com`);
const newEmailAddress = new EmailAddress(`johndoe${randomSuffix}@example.com`);

await setupEmailRegistry(emailRegistry);
const user = userFactory.createUser({id, username, role,
    password: createMd5Hash(password), emailAddress});
await userRepository.save(user);
const savedUser = await userRepository.load(id);
savedUser.emailAddress = newEmailAddress;
await userRepository.save(savedUser);
console.log(await userRepository.findUserByEmail(newEmailAddress));
```

Task board context

Compared to the other parts, the task board context implementation requires the most complex changes and extensions for enabling persistence. While the type `TaskSummary` remains unmodified and the class `TaskBoard` only requires minor adjustments, the component `Task` is changed drastically. As other modified Entity types, its constructor removes the Event Bus argument and introduces an event collection. Also, it defines the parameters `status`, `assigneeId` and `isExistingTask` to support Entity reconstitution. The function `setStatus()` creates an event with the type “`TaskStatusChanged`” and adds it to the collection of new

events. For existing tasks, the status field is assigned directly. For both Entity types, a Repository component is implemented, again as subclass of `ConcurrencySafeFilesystemRepository`. Finally, the state synchronization component is adapted to correctly work with persistence.

The first examples show the reworked constructor and the status setter of the task Entity:

Task board context: Task constructor

```
constructor(
    {id, title, description = '', status = 'todo', assigneeId, isExistingTask}) {
    verify('valid id', id != null);
    Object.defineProperty(this, 'id', {value: id, writable: false});
    Object.assign(this, {title, description, assigneeId});
    if (isExistingTask) this.#status = status;
    else this.status = status;
}
```

Task board context: Task status setter

```
set status(status) {
    verify('valid status', validStatus.includes(status));
    verify('active task assignee', status !== 'in progress' || this.assigneeId);
    this.#status = status;
    this.#newDomainEvents.push(
        new TaskStatusChangedEvent({taskId: this.id, status}));
}
```

The following code shows the refactored version of the task board constructor that accepts initial task summaries for Entity reconstitution:

Task board context: Task board constructor

```
constructor({id, initialTaskSummaries = []}) {
    verify('valid id', id != null);
    Object.defineProperty(this, 'id', {value: id, writable: false});
    initialTaskSummaries.forEach(taskSummary => this.addTask(taskSummary));
}
```

The next examples provide the Repository classes `TaskRepository` and `TaskBoardRepository`:

Task board context: Task Repository

```
class TaskRepository extends ConcurrencySafeFilesystemRepository {

    constructor({storageDirectory}) {
        super({
            storageDirectory,
            convertToData: ({id, title, description, status, assigneeId}) =>
                ({id, title, description, status, assigneeId}),
            convertToEntity: ({id, title, description, status, assigneeId}) => new Task(
                {id, title, description, status, assigneeId, isExistingTask: true}),
        });
    }

    findTasksByAssigneeId(assigneeId) {
        return this.loadAll()
            .then(tasks => tasks.filter(task => task.assigneeId === assigneeId));
    }
}
```

Task board context: Task board Repository

```
class TaskBoardRepository extends ConcurrencySafeFilesystemRepository {

    constructor({storageDirectory}) {
        const convertToData = entity => {
            const taskSummariesData = entity.getTasks().map(
                taskSummary => ({taskId: taskSummary.taskId, status: taskSummary.status}));
            return {id: entity.id, taskSummaries: taskSummariesData};
        };
        const convertToEntity = data => {
            const taskSummaries = data.taskSummaries.map(
                data => new TaskSummary({taskId: data.taskId, status: data.status}));
            return new TaskBoard({id: data.id, initialTaskSummaries: taskSummaries});
        };
        super({storageDirectory, convertToData, convertToEntity});
    }

    async findTaskBoardByTaskId(taskId) {
        const taskBoards = await this.loadAll();
        return taskBoards.find(taskBoard => taskBoard.containsTask(taskId));
    }
}
```

}

The components `TaskRepository` and `TaskBoardRepository` define specialized converter functions and pass them to their parent constructor. While the task conversion primarily ensures a correct type, reconstituting task board Entities also requires the instantiation of `TaskSummary` Value Objects. In addition to their inherited generic functionality, each of the Repository components defines a custom query. The class `TaskBoardRepository` implements the function `findTaskBoardByTaskId()` to find the board a task is referenced from. For determining all tasks that are assigned to a specific team member, the `TaskRepository` component provides the operation `findTasksByAssigneeId()`. Both queries are used for enabling the synchronization mechanism to correctly work with persistence.

The next code shows a refactored version of the task board synchronization component:

Task board context: Task board synchronization

```
const taskBoardSynchronization = {

    activate({taskBoardRepository, taskRepository, eventBus}) {
        eventBus.subscribe(TaskStatusChangedEvent.type, async ({data}) => {
            const taskBoard =
                await taskBoardRepository.findTaskBoardByTaskId(data.taskId);
            if (!taskBoard) return;
            taskBoard.updateTaskStatus(data.taskId, data.status);
            await taskBoardRepository.save(taskBoard);
        });
        eventBus.subscribe('TeamMemberRemovedFromTeam', async ({data}) => {
            const tasks = await taskRepository.findTasksByAssigneeId(data.teamMemberId);
            return Promise.all(tasks.map(task => {
                if (task.status === 'in progress') task.status = 'todo';
                task.assigneeId = undefined;
                return taskRepository.save(task);
            }));
        });
    },
};
```

The reworked synchronization mechanism correctly works with persistent components and improves the original functionality. Both previously separated parts are combined into the

command `activate()`. As arguments, the function expects an Event Bus and Repositories for both tasks and task boards. The subscriber for the Domain Event “TaskStatusChanged” correctly works for all persisted boards. For every status change, it retrieves the affected Entity via `findTaskBoardByTaskId()`, updates the according task summary and saves the result. Executing `containsTask()` is unnecessary, as the returned board is inherently the correct one. The subscriber for the Domain Event “TeamMemberRemoved” determines all tasks of an assignee by executing `findTasksByAssigneeId()`. This replaces the manual search through a custom in-memory database. For every returned Entity, the assignee is removed and the change is saved.

The last example illustrates the usage of the task board context with persistence ([run code usage](#)):

Task board context: Overall usage

```
const taskRepository = new TaskRepository({storageDirectory: taskStorageDirectory});
const taskBoardRepository = new TaskBoardRepository(
  {storageDirectory: taskBoardStorageDirectory});

const publisher = new DomainEventPublisher({repository: taskRepository, eventBus,
  publishedEventIdsDirectory: `${taskStorageDirectory}/published-event-ids`});
taskBoardSynchronization.activate({eventBus, taskBoardRepository, taskRepository});
publisher.activate();

const loadTaskBoardAndLogTasks = async taskBoardId =>
  console.log((await taskBoardRepository.load(taskBoardId)).getTasks());

const [taskBoardId, taskId, teamMemberId] =
  [generateId(), generateId(), generateId()];

let taskBoard = new TaskBoard({id: taskBoardId});
await taskBoardRepository.save(taskBoard);
let task = new Task({id: taskId, title: 'write tests'});
task.assigneeId = teamMemberId;
task.description = 'write unit tests for new feature';
await taskRepository.save(task);
[taskBoard, task] = await Promise.all([
  taskBoardRepository.load(taskBoardId), taskRepository.load(taskId)]);
const taskSummary = TaskSummary.createFromTask(task);
taskBoard.addTask(taskSummary);
await taskBoardRepository.save(taskBoard);
await loadTaskBoardAndLogTasks(taskBoardId);
task = await taskRepository.load(taskId);
task.status = 'in progress';
```

```
await taskRepository.save(task);
await timeout(125);
await loadTaskBoardAndLogTasks(taskBoardId);
await eventBus.publish(
    new TeamMemberRemovedFromTeamEvent({teamId: generateId(), teamMemberId}));
await timeout(125);
await loadTaskBoardAndLogTasks(taskBoardId);
```

Queries instead of state replication

Instead of replicating and synchronizing task state, the statuses of all tasks on a board could be determined via Repositories. While it is an essential functionality, it is not necessarily the responsibility of the `TaskBoard` Entity type. The behavior could be extracted, as it is read-only and therefore not involved in any invariant. This approach would require to load a task board and then to retrieve all Entities for the referenced task identifiers. Ultimately, it could eliminate the need for state synchronization. However, the complexity would only be shifted. Furthermore, the result would still be eventually consistent, as it spans across multiple transactional boundaries. This design issue is solved automatically when introducing CQRS. Therefore, the code requires currently no refactoring.

The introduction of Repositories to the Sample Application enables a persistent Domain Model. The next step is to introduce the Application Layer of the software.

Chapter 10: Application Services

The Application Layer of a software is most commonly implemented as a set of **Application Services**. For this architectural part, the responsibilities are to execute use cases, manage transactions and handle security. Typically, this involves utilizing infrastructural functionalities for persistence, and accessing Domain Model components for behavior execution. Furthermore, it can also include the orchestration of interactions between multiple related parts. In general, the Application Layer must not contain behavior or maintain state that belongs to the Domain Model. However, holding technical information such as references to dependencies is acceptable. Seen from the outside, Application Services shield the Domain part and act as controlling facade. Every individual service should be responsible for a single use case without drawing dependencies to others.

Service design

There are multiple options for the design and the implementation of Application Services. [Vernon, p. 143] describes three different implementation styles. All of them share the principle that every use case is expressed as individual action. The “dedicated style” implements each service as separate structure with one main function. Ideally, the name of this operation is identical across components. The “categorized style” gathers multiple related Application Services into one common structure and exposes each of them as individual operation. While the first approach produces small units with single responsibilities, the second variant is typically easier to understand. Finally, the “messaging style” expects to receive service execution requests as asynchronous notifications. As this approach is more of an infrastructural topic, it is not further discussed at this point.

The following code shows an abstract example for dedicated Application Service classes:

Application Services: Dedicated style

```
class FirstUseCaseService {
  execute() { /* .. */ }
}

class SecondUseCaseService {
  execute() { /* .. */ }
}

const firstUseCaseService = new FirstUseCaseService();
const secondUseCaseService = new SecondUseCaseService();
firstUseCaseService.execute();
secondUseCaseService.execute();
```

The next example illustrates the categorized equivalent of the same functionality:

Application Services: Categorized style

```
class ApplicationServices {
  firstUseCase() { /* .. */ }
  secondUseCase() { /* .. */ }
}

const applicationServices = new ApplicationServices();
applicationServices.firstUseCase();
applicationServices.secondUseCase();
```



What about standalone functions?

Application Services can also be implemented as free functions. Their enclosure in stateful objects is motivated by the fact that they commonly have dependencies to other components. This approach enables the required references to be saved as fields inside the containing structure. When aiming for a more functional approach, dependencies can be bound early to standalone operations via partial application.

Independent of implementation styles, there are some general design recommendations. For one, Application Services should have intention-revealing names that conform to the respective Ubiquitous Language. One useful pattern is to combine the name of a targeted component and its behavior to execute. For example, an operation that invokes the

command `login()` on user Entities can be called `loginUser()`. Application Services should always be treated as asynchronous, as they typically incorporate persistence or network communication. Furthermore, they should not actively generate identifiers but rather receive them as arguments. This makes their behavior more predictable and enables idempotent procedures. Finally, all services should either use simple types or specialized data structures as parameters and return values. This allows consumers to stay independent of domain-specific details.

Exemplary Application Service names

Component	Action	Potential service name
browser tab	reload()	reloadBrowserTab()
meeting	addParticipant()	addMeetingParticipant()
guestbook	writeMessage()	writeGuestbookMessage()
user	login()	loginUser()
newsletter	subscribe()	addNewsletterSubscription()

Example: Notepad

Consider implementing the Domain Model, the Repositories and the Application Services for a notepad software. The goal is to be able to create and update text notes. Each note incorporates a title and arbitrary content. Apart from creating new instances, the main required functionality is to update both of the attributes individually. This requires them to be mutable. Also, notes need to be distinguishable independent of their content. Consequently, they must be implemented as Entity type with a conceptual identity. Other than with previous example implementations, the Application Layer is also required in this case. Consumers must be able to create notes, update their attributes and to retrieve the data of specific instances. All this must be possible without directly depending on the Domain Layer.

The first code shows the note Entity implementation ([run code usage](#)):

Notepad: Note Entity

```
class Note {  
  id; title; content;  
  
  constructor({id, title = '', content = ''}) {  
    Object.defineProperty(this, 'id', {value: id, writable: false});  
    Object.assign(this, {title, content});  
  }  
}
```

The second code provides a specialized data structure to be used as return value from Application Services ([run code usage](#)):

Notepad: Note values

```
const NoteValues = function({title, content}) {  
  Object.freeze(Object.assign(this, {title, content}));  
};  
  
NoteValues.createFromNote = note =>  
  new NoteValues({title: note.title, content: note.content});
```

The next example shows the implementation of the actual Application Services component:

Notepad: Application Services

```
class NoteApplicationServices {  
  
  #noteRepository;  
  
  constructor({noteRepository}) {  
    this.#noteRepository = noteRepository;  
  }  
  
  async createNote({noteId, content, category}) {  
    const note = new Note({id: noteId, content, category});  
    await this.#noteRepository.save(note);  
  }  
  
  async updateNoteContent({noteId, content}) {
```

```
const note = await this.#noteRepository.load(noteId);
note.content = content;
await this.#noteRepository.save(note);
}

async updateNoteTitle({noteId, title}) {
  const note = await this.#noteRepository.load(noteId);
  note.title = title;
  await this.#noteRepository.save(note);
}

async getNote({noteId}) {
  return NoteValues.createFromNote(await this.#noteRepository.load(noteId));
}

}
```

The component `NoteValues` represents an immutable data structure that can be used as return value from the Application Services. With the help of its static factory function `createFromNote()`, an instance can be created out of an existing note Entity. The class `NoteApplicationServices` implements the actual use cases. Creating a new item is done via the function `createNote()`. Both the commands `updateNoteTitle()` and `updateNoteContent()` are responsible for modifying the state of an existing Entity. Finally, the query `getNote()` retrieves and returns the current title and content of a note. Overall, the service function names adhere to the general naming recommendation. Also, every operation is working asynchronously. The class directly depends on Domain components. In contrast, the Repository is referenced as abstract functionality and injected at runtime.

The final example shows an exemplary usage of the previously illustrated components ([run code](#)):

Notepad: Usage

```
const noteRepository = new FilesystemRepository({storageDirectory,
  convertToData: entity => entity, convertToEntity: data => new Note(data)});

const noteApplicationServices = new NoteApplicationServices({noteRepository});

const noteId = generateId();
await noteApplicationServices.createNote(
  {noteId, title: 'work goals', content: 'realize own projects'});
await noteApplicationServices.updateNoteContent(
```

```
{noteId, content: 'realize own projects, do more sports'});
await noteApplicationServices.updateNoteTitle({noteId, title: 'work & life'});
console.log(await noteApplicationServices.getNote({noteId}));
```

The code starts with creating an object of the `FilesystemRepository` component with according converter functions. Then, the class `NoteApplicationServices` is instantiated with the `Repository` as constructor argument. This is followed by the actual usage example. As preliminary step, an Entity identifier is created. Next, the service `createNote()` is executed with the identifier, a title and sample content as arguments. This code illustrates the approach of passing in client-originated identities to the Application Layer. Afterwards, the commands `updateNoteContent()` and `updateNoteTitle()` are executed to update the example note. Finally, the current state is retrieved via the query `getNote()` and output to the console. Amongst other things, this example demonstrates that consumers of Application Services can stay completely independent of an underlying Domain part.

Use case scenarios

There are different types of use case scenarios executed by Application Services. The standard procedure consists of loading an Entity, executing behavior and saving the resulting change. Another variant of this is when creating a new instance, in which case the loading is obsolete. The third typical scenario is to access a component, optionally execute a computation and return information. All of these flows are covered by the previously illustrated example. On top of that, there are more atypical scenarios. One is to load multiple Entities, orchestrate an interaction between them and either save the changes or return the results. Another use case type is when multiple actions are performed on a single component. Furthermore, it is also possible to exclusively execute stateless computations.

Example: Notepad extensions

Consider extending the Domain Model implementation and the Application Services of the notepad software. As an addition to the previously implemented behavior, there are two more required functionalities. The first one is to update both the title and the content of a note within a single transaction. Although it is generally better to perform individual updates, changing these attributes simultaneously represents a common use case. The second required functionality is to generate a note title based on a given text. For this purpose, the Domain Model defines a generic mechanism to extract the first three words of an arbitrary text. This

functionality can be used from outside the Application Layer whenever a user does not define a custom title value.

The first example shows the service operation for generating a title based on given content:

Notepad: Generate title function

```
const generateNoteTitle = content => content.split(' ').slice(0, 3).join(' ');
```

The next code implements the extended functionalities in the Application Services class ([run code usage](#)):

Notepad: Application Services extensions

```
async updateNoteTitleAndContent({noteId, title, content}) {
    const note = await this.#noteRepository.load(noteId);
    note.title = title;
    note.content = content;
    await this.#noteRepository.save(note);
}

async generateNoteTitle({content}) { return generateNoteTitle(content); }
```

The stateless operation `generateNoteTitle()` generates a note title by extracting the first three words out of an arbitrary text. Whenever a provided input contains less than three words, all of them are returned again. The class `NoteApplicationServices` is extended with two use case operations. Updating both the title and the content of a note is done via the command `updateNoteTitleAndContent()`. This service operation loads an Entity, mutates the attributes `title` and `content`, and saves the resulting changes. The query `generateNoteTitle()` creates a title by calling the previously introduced service function. Both of the additional use cases represent an atypical scenario for an Application Service as described previously.

Transactions and Processes

The execution of an individual Application Service should ideally only affect a single transaction. With regard to a Domain Model implementation, this means to not create or modify more than one Aggregate. The goal is to perform an atomic transition from one valid state to another. There must be no possibility for an in-between interruption that leads to a corrupted state, be it temporarily or permanently. Also, non-essential secondary

functionalities, such as informational logging, should be executed in a deferred way. This is because such aspects must not affect the success of the actual behavior. Despite the aim for transactional services, there are scenarios that require state changes across multiple consistency boundaries. The challenge is to facilitate them in an adequate and resilient way.

Processes

Every use case that incorporates multiple transactional steps is a so-called **Process**. The implementation of such a procedure can be done in different ways. One option is to execute everything within a single service. While this approach has a low complexity, it is only feasible when an in-between interruption cannot produce corrupted states. The second possibility is to expose standalone services for each step of a process. This method makes sense when the orchestration is the responsibility of the consumer. Another option is to connect individual steps via event notifications. With this approach, a single entry point is exposed that initiates the process by executing its first step. The consequential state change triggers an asynchronous event notification, which causes the process to advance further.

Process implementation approaches

Approach	Advantage	Disadvantage
Single Service	Low complexity	Interruption leads to in-between states
Individual Services	Low complexity	Responsibility is shifted to consumers
Event notifications	High resilience	Additional complexity & infrastructure



Orchestration vs. Choreography

Processes can be managed in two different ways. They can be orchestrated through a centralized **Process Manager** that controls and tracks their progress. Alternatively, multiple individual parts can perform a **Choreography**, where each reacts to a trigger and performs a single step. Typically, the Orchestration approach makes most sense for complex processes with many steps or non-linear paths.

Example: Classified ads platform

Consider implementing the Domain Model, the Repositories and the Application Services for a simplified version of a classified ads platform. This is the same topic as it is used for

the example in Chapter 7, section “Distribution and processing”. Users can create classified ads for arbitrary goods and potential buyers can contact them. The business model is to offer a free basic version and to provide additional paid services on top. By default, every user is allowed to post up to three ads. This limit can be increased through the payment of a certain fee. The example focuses on implementing the free variant together with the classified ad number restriction. However, the actual upgrade functionality is excluded to avoid unnecessary complexity.

The relevant Domain Model part consists of two components. One is the user account, which contains an immutable identifier, an e-mail address and a list of classified ad references. The second part is the classified ad, which encloses attributes for identifier, seller identity, title, description and price. Both the identifier and the reference to the seller must be constant values. For simplicity reasons, the price is implemented as plain numerical value. The required functionalities are to create user accounts and to place classified ads, which itself consists of multiple individual steps. First, an ad Entity must be created. Then, its identity must be added to the collection of the respective user account. Finally, a confirmation message should be triggered to report the operational success.

The first two examples provide the Entity type for the user account and the classified ad ([run code usage](#)):

Classified ads platform: User account Entity

```
class UserAccount {  
  
    id; emailAddress; #classifiedAdIds;  
  
    constructor({id, emailAddress, classifiedAdIds = []}) {  
        if (classifiedAdIds.length > 3) throw new Error('not allowed');  
        Object.defineProperty(this, 'id', {value: id, writable: false});  
        Object.defineProperty(this, 'classifiedAds',  
            {get: () => this.#classifiedAdIds.slice(), enumerable: true});  
        this.#classifiedAdIds = classifiedAdIds;  
        this.emailAddress = emailAddress;  
    }  
  
    addClassifiedAd(classifiedAdId) {  
        if (!this.canCreateClassifiedAd()) throw new Error('not allowed');  
        this.#classifiedAdIds.push(classifiedAdId);  
    }  
  
    canCreateClassifiedAd() { return this.#classifiedAdIds.length < 3; }  
}
```

```
}
```

Classified ads platform: Classified ad Entity

```
class ClassifiedAd {  
  
    id; sellerId; title; description; price;  
  
    constructor({id, sellerId, title, description = '', price}) {  
        Object.defineProperty(this, 'id', {value: id, writable: false});  
        Object.assign(this, {sellerId, title, description, price});  
    }  
  
}
```

The third example shows the initial implementation of the Application Services component:

Classified ads platform: Application Services with multiple transactions

```
class ApplicationServices {  
  
    #userAccountRepository; #classifiedAdRepository;  
  
    constructor({userAccountRepository, classifiedAdRepository}) {  
        this.#userAccountRepository = userAccountRepository;  
        this.#classifiedAdRepository = classifiedAdRepository;  
    }  
  
    async createUserAccount({userAccountId, emailAddress}) {  
        const userAccount = new UserAccount({id: userAccountId, emailAddress});  
        await this.#userAccountRepository.save(userAccount);  
    }  
  
    async createClassifiedAd({classifiedAdId, sellerId, title, description, price}) {  
        const userAccount = await this.#userAccountRepository.load(sellerId);  
        if (!userAccount.canCreateClassifiedAd()) throw new Error('not allowed');  
        const classifiedAd = new ClassifiedAd(  
            {id: classifiedAdId, sellerId, title, description, price});  
        await this.#classifiedAdRepository.save(classifiedAd);  
        userAccount.addClassifiedAd(classifiedAdId);  
        await this.#userAccountRepository.save(userAccount);  
        console.log(`classified ad ${classifiedAdId} published by ${userAccount.id}`);  
    }  
}
```

}

The component `UserAccount` expresses the concept of an individual user account. For referencing classified ads, the class defines a collection of identifiers. Adding a new entry is done via the command `addClassifiedAd()`. Determining whether a user account is below the allowed limit is achieved with the query `canCreateClassifiedAd()`. The component `ClassifiedAd` represents the classified ad component. Apart from the ability to modify its enclosed attributes, it does not expose any specialized behavior. Finally, the class `ApplicationServices` implements the actual use cases. Its command `createUserAccount()` has the sole responsibility to create and save a user account. In contrast, the function `createClassifiedAd()` performs all the steps of the process for creating a classified ad. This means that the operation affects two separate transactions and one secondary functionality.

The next example shows a usage of the previously implemented components ([run code](#)):

Classified ads platform: Usage with multiple transactions

```
const userAccountRepository = new FilesystemRepository({
  storageDirectory: userAccountStorageDirectory,
  convertToData: entity => entity, convertToEntity: data => new UserAccount(data),
});

const classifiedAdRepository = new FilesystemRepository({
  storageDirectory: classifiedAdStorageDirectory,
  convertToData: entity => entity, convertToEntity: data => new ClassifiedAd(data),
});

const applicationServices =
  new ApplicationServices({userAccountRepository, classifiedAdRepository});

const userAccountId = generateId(), classifiedAdId = generateId();

await applicationServices.createUserAccount(
  {userAccountId, emailAddress: 'test@example.com'});

const interval = setInterval(logPersistedEntities);

await applicationServices.createClassifiedAd(
  {classifiedAdId, sellerId: userAccountId, title: 'Some ad', price: 1000});

clearInterval(interval);
logPersistedEntities();
```

The code starts with creating one Repository for user accounts and one for classified ads. Next, the class `ApplicationServices` is instantiated with the Repositories as arguments. This is followed by defining two identifiers. Afterwards, the service `createUserAccount()` is executed. Then, an interval is created to periodically log all persisted Entities using the helper `logPersistedEntities()`. As next step, the service `createClassifiedAd()` is invoked. After its successful completion, the interval is stopped. Finally, the state of both Entities is logged again. Executing the code eventually produces an output where a classified ad exists without being referenced from a user account. This circumstance itself merely demonstrates the Eventual Consistency of the process. The actual issue is that a service interruption can cause this in-between state to exist permanently.

The next example provides a modified version of the Application Services, broken down into individual steps:

Classified ads platform: Application Services with individual steps

```
async createClassifiedAd({classifiedAdId, sellerId, title, description, price}) {
    const userAccount = await this.#userAccountRepository.load(sellerId);
    if (!userAccount.canCreateClassifiedAd()) throw new Error('not allowed');
    const classifiedAd = new ClassifiedAd({
        id: classifiedAdId, sellerId, title, description, price});
    await this.#classifiedAdRepository.save(classifiedAd);
}

async addClassifiedAdToUserAccount({userAccountId, classifiedAdId}) {
    const userAccount = await this.#userAccountRepository.load(userAccountId);
    userAccount.addClassifiedAd(classifiedAdId);
    await this.#userAccountRepository.save(userAccount);
}

sendClassifiedAdConfirmation({userAccountId, classifiedAdId}) {
    console.log(`classified ad ${classifiedAdId} published by ${userAccountId}`);
}
```

The following code shows an exemplary usage of the refactored Application Services ([run code usage](#)):

Classified ads platform: Usage with individual steps

```
const userAccountId = generateId(), classifiedAdId = generateId();
await applicationServices.createUserAccount(
  {userAccountId, emailAddress: 'test@example.com'});
await applicationServices.createClassifiedAd(
  {classifiedAdId, sellerId: userAccountId, title: 'Some ad', price: 1000});
await applicationServices.addClassifiedAdToUserAccount(
  {userAccountId, classifiedAdId});
await applicationServices.sendClassifiedAdConfirmation(
  {userAccountId, classifiedAdId});
```

The reworked class `ApplicationServices` defines a separate service for each of the steps involved in creating a classified ad. Instantiating and persisting the actual Entity is done with the operation `createClassifiedAd()`. The command `addClassifiedAdToUserAccount()` is responsible for adding an ad identifier to the collection of a user account. Finally, the function `sendClassifiedAdConfirmation()` outputs a confirmation message. Each operation is an individual service that only affects a single transaction. The refactored usage code demonstrates the increased complexity on the consumer side. All three steps must be executed sequentially in a specific order. The Application Layer fails to express and enforce the process of creating a classified ad. Also, similar to the first approach, an interruption on the consumer side can lead to a permanently corrupted state.

As preparation for the third approach, the first examples introduce Domain Events to the Entities ([run code usage](#)):

Classified ads platform: Domain Events

```
const ClassifiedAdCreatedEvent = createEventType('ClassifiedAdCreated', {
  classifiedAdId: 'string', sellerId: 'string', title: 'string',
  description: 'string', price: 'number',
});

const ClassifiedAdAddedToUserAccountEvent = createEventType(
  'ClassifiedAdAddedToUserAccount',
  {userAccountId: 'string', classifiedAdId: 'string'},
);
```

Classified ads platform: Classified ad Entity with Domain Event

```
class ClassifiedAd {  
  
    id; sellerId; title; description; price; #newDomainEvents = [];  
  
    constructor({id, sellerId, title, description = '', price, isNewAd = true}) {  
        Object.defineProperty(this, 'id', {value: id, writable: false});  
        Object.assign(this, {sellerId, title, description, price});  
        if (isNewAd) this.#newDomainEvents.push(new ClassifiedAdCreatedEvent(  
            {classifiedAdId: this.id, sellerId, title, description, price}));  
    }  
  
    get newDomainEvents() { return this.#newDomainEvents; }  
  
}
```

Classified ads platform: User account Entity with Domain Event

```
class UserAccount {  
  
    id; emailAddress; #classifiedAdIds; #newDomainEvents = [];  
  
    constructor({id, emailAddress, classifiedAdIds = []}) {  
        if (classifiedAdIds.length > 3) throw new Error('not allowed');  
        Object.defineProperty(this, 'id', {value: id, writable: false});  
        Object.defineProperty(this, 'classifiedAds',  
            {get: () => this.#classifiedAdIds.slice(), enumerable: true});  
        this.#classifiedAdIds = classifiedAdIds;  
        this.emailAddress = emailAddress;  
    }  
  
    addClassifiedAd(classifiedAdId) {  
        if (!this.canCreateClassifiedAd()) throw new Error('not allowed');  
        this.#classifiedAdIds.push(classifiedAdId);  
        this.#newDomainEvents.push(new ClassifiedAdAddedToUserAccountEvent(  
            {userAccountId: this.id, classifiedAdId}));  
    }  
  
    canCreateClassifiedAd() { return this.#classifiedAdIds.length < 3; }  
  
    get newDomainEvents() { return this.#newDomainEvents.slice(); }  
  
}
```

The Domain Event types “ClassifiedAdCreated” and “ClassifiedAdAddedToUserAccount” define the two domain-specific occurrences that are part of the ad creation process. The Entity class `ClassifiedAd` is extended with multiple aspects. For one, it accepts the additional constructor argument `isNewAd` to differentiate between creation and reconstitution. Secondly, it defines the attribute `#newDomainEvents` and an according public accessor. Upon creating a new instance, an event of the type “`ClassifiedAdCreated`” is created and added to the collection. The user account Entity type is also extended with a collection for new Domain Events. Executing its command `addClassifiedAd()` creates an event of the type “`ClassifiedAdAddedToUserAccount`” and adds it to the internal event collection.

The next code shows another reworked implementation of the Application Services class:

Classified ads platform: Application Services with Eventual Consistency

```
class ApplicationServices {  
  
    #userAccountRepository; #classifiedAdRepository;  
  
    constructor({userAccountRepository, classifiedAdRepository}) {  
        this.#userAccountRepository = userAccountRepository;  
        this.#classifiedAdRepository = classifiedAdRepository;  
    }  
  
    async createUserAccount({userAccountId, emailAddress}) {  
        await this.#userAccountRepository.save(  
            new UserAccount({id: userAccountId, emailAddress}));  
    }  
  
    async createClassifiedAd({classifiedAdId, sellerId, title, description, price}) {  
        const userAccount = await this.#userAccountRepository.load(sellerId);  
        if (!userAccount.canCreateClassifiedAd()) throw new Error('not allowed');  
        await this.#classifiedAdRepository.save(new ClassifiedAd(  
            {id: classifiedAdId, sellerId, title, description, price, isNewAd: true}));  
    }  
}
```

This is complemented with an event handlers component that connects the steps of the ad creation process via event notifications:

Classified ads platform: Domain Event handlers

```
class DomainEventHandlers {  
  
    #userAccountRepository; #eventBus;  
  
    constructor({userAccountRepository, eventBus}) {  
        this.#userAccountRepository = userAccountRepository;  
        this.#eventBus = eventBus;  
    }  
  
    activate() {  
        this.#eventBus.subscribe(ClassifiedAdCreatedEvent.type, async event => {  
            const {data: {classifiedAdId, sellerId}} = event;  
            const userAccount = await this.#userAccountRepository.load(sellerId);  
            userAccount.addClassifiedAd(classifiedAdId);  
            await this.#userAccountRepository.save(userAccount);  
        });  
        this.#eventBus.subscribe(ClassifiedAdAddedToUserAccountEvent.type, event => {  
            const {data: {userAccountId, classifiedAdId}} = event;  
            console.log(`classified ad ${classifiedAdId} published by ${userAccountId}`);  
        });  
    }  
}
```

The reworked class `ApplicationServices` is exclusively responsible for creating user accounts and classified ads. Both of the use cases only affect a single transaction. The previously introduced functions `addClassifiedAdToUserAccount()` and `sendClassifiedAdConfirmation()` are removed completely. Instead, the class `DomainEventHandlers` is responsible for subscribing to Domain Event notifications and executing corresponding consequential actions. Its constructor expects a Repository for user accounts and an Event Bus instance. Executing the command `activate()` registers two subscribers. For every event with the type “`ClassifiedAdCreated`”, the respective classified ad identifier is added to the affected user account. This is done by loading the corresponding Entity, executing its operation `addClassifiedAd()` and persisting the changes. Each “`ClassifiedAdAddedToUserAccount`” event causes to log an exemplary notification message to the console.

Finally, the next code demonstrates the use of the refactored implementation ([run code](#)):

Classified ads platform: Usage with Eventual Consistency

```
const userAccountRepository = new EventPublishingFilesystemRepository({
  storageDirectory: `${rootStorageDirectory}/user-account`, eventBus,
  convertToData: entity => ({...entity}),
  convertToEntity: data => new UserAccount(data),
});

const classifiedAdRepository = new EventPublishingFilesystemRepository({
  storageDirectory: `${rootStorageDirectory}/classified-ad`, eventBus,
  convertToData: entity => ({...entity}),
  convertToEntity: data => new ClassifiedAd({...data, isNewAd: false}),
});

const applicationServices = new ApplicationServices(
  {userAccountRepository, classifiedAdRepository});
const eventHandlers = new DomainEventHandlers({userAccountRepository, eventBus});
eventHandlers.activate();

const userAccountId = generateId(), classifiedAdId = generateId();
await applicationServices.createUserAccount(
  {userAccountId, emailAddress: 'test@example.com'});
await applicationServices.createClassifiedAd(
  {classifiedAdId, sellerId: userAccountId, title: 'Some ad', price: 1000});
```

The code starts with creating one Repository for user accounts and one for classified ads using the class `EventPublishingFilesystemRepository`. While it is not recommended to publish Domain Events directly from a Repository, it avoids additional complexity for the example. Afterwards, the class `ApplicationServices` is instantiated. Also, an instance of the component `DomainEventHandlers` is created and its command `activate()` is invoked. Finally, the services `createUserAccount()` and `createClassifiedAd()` are executed. The code looks similar to the first approach. Even more, the system is also temporarily accessible in an in-between state. The difference is that the ad creation is implemented as a chain of asynchronous steps with a transactional entry point. Furthermore, the risk of permanent state corruption can be eliminated through a guaranteed event distribution.



What about consistency delays?

The example implementation establishes Eventual Consistency between the actual count of classified ads and the associated number per user account. Consequently, a delay in consistency can cause to allow an ad creation despite a reached limitation. This issue can be mitigated by extending the subscriber function for the “`ClassifiedAdCreated`” event to conditionally perform a corrective action.

Cross-cutting concerns

In addition to its main responsibility, the Application Layer commonly also manages generic aspects that spread across multiple use cases. These cross-cutting concerns are secondary functionalities that belong either to the Application part itself or to the Infrastructure Layer. Typical examples are security, validation, error detection, caching and logging. Directly integrating such aspects into multiple individual services can cause them to scatter across a software. Keeping the Application Layer clean and free from concrete dependencies can be achieved through a unified and abstract integration. There are two common design patterns that help with implementing this approach. Both the concepts Decorator and Middleware enable to enrich or modify existing functionality in an abstract way. Although they have slightly different characteristics, the patterns aim for the same goal.

The following code provides a function to decorate another function:

Decoration: Decorate function utility

```
const decorateFunction = (originalFunction, decorator) =>
  (...args) => decorator(originalFunction, args);
```

The operation `decorateFunction()` expects a function and a decorator as parameters. It returns another function that delegates each invocation to the decorator, passing in the original operation and the respective arguments. The implementation provides a number of possibilities. For one, it enables to integrate additional aspects before and after the execution of the original. Also, the respective arguments of individual calls can be modified or extended. Furthermore, it is possible to decide whether to execute the original at all. These capabilities cause the decorator signature to be moderately complex. There are other implementation approaches that aim for a simpler interface. However, they implicitly also reduce the possibilities. The shown utility can be used to decorate an operation with virtually any aspect, regardless of details.

The next example shows a usage of the decorator utility ([run code](#)):

Decoration: Decorate function utility usage

```
const sum = (a, b) => a + b;

const loggingDecorator = (originalFunction, args) => {
    console.log(`executing ${originalFunction.name} with args ${args.join(', ')}`);
    const result = originalFunction(...args);
    console.log(`result: ${result}`);
    return result;
};

const sumWithLogging = decorateFunction(sum, loggingDecorator);

sumWithLogging(3, 5);

const validationDecorator = (originalFunction, args) => {
    if (args.some(input => typeof input != 'number'))
        throw new TypeError('invalid argument');
    return originalFunction(...args);
};

const sumWithLoggingAndValidation =
    decorateFunction(sumWithLogging, validationDecorator);

sumWithLoggingAndValidation(1, 'a');
```

The code starts with defining the mathematical computation `sum()`. Next, it creates the decorator `loggingDecorator()`, which consists of multiple steps. First, it logs the name of a function and the provided arguments. Then, it executes the original and logs the result. Finally, the value is returned to the caller. The `sum()` calculation is decorated by invoking `decorateFunction()` and passing in the logging mechanism. After an exemplary execution of `sumWithLogging()`, the code defines the operation `validationDecorator()`. This decorator ensures that all given arguments are numbers before executing the original. If a parameter has another type, it throws an exception. The original calculation is further extended with the validation aspect. As last step, the operation `sumWithLoggingAndValidation()` is executed. This example illustrates various scenarios for the helper `decorateFunction()`.

The next code combines the Middleware pattern with a Proxy for adding cross-cutting concerns to all functions of an object:

Decoration: Middleware Proxy

```
const MiddlewareProxy = function(target) {
  const executeOriginal =
    ({originalFunction, args}) => originalFunction.apply(target, args);

  let middlewareChain = executeOriginal;

  const get = (target, property) => {
    if (property === 'addMiddleware') return addMiddleware;
    const value = target[property];
    if (typeof value !== 'function') return value;
    return (...args) => middlewareChain({originalFunction: value, args});
  };

  const addMiddleware = middleware => {
    const next = middlewareChain;
    middlewareChain =
      ({originalFunction, args}) => middleware({originalFunction, args, next});
  };

  return new Proxy(target, {get});
};
```

The class `MiddlewareProxy` provides the possibility to extend all functions of an object with additional aspects. As constructor argument, it expects an object to decorate. The returned proxy intercepts every property access and checks for existence on the original. If the property name points to a value, it is simply returned. In case of a function, the component yields an operation that executes its middleware chain. Initially, this chain only consists of the operation `executeOriginal()`, which invokes the original function. Adding new middleware is done via the command `addMiddleware()`. This operation expects a callback as argument that accepts the parameters `originalFunction`, `args` and `next`. Upon each invocation, the `next()` parameter is set to the respectively next function item in the middleware chain.

**Why not subclass `Proxy`?**

The component `MiddlewareProxy` is not implemented as a subclass of `Proxy` because this is technically not possible in JavaScript.

The following example illustrates the usage of the middleware proxy component ([run code](#)):

Decoration: Middleware proxy usage

```
const calculator = {
  storedValue: 42,
  sum: (a, b) => a + b,
  product: (a, b) => a * b,
};

const calculatorWithMiddleware = new MiddlewareProxy(calculator);

const loggingMiddleware = ({originalFunction, args, next}) => {
  console.log(`execute: ${originalFunction.name}`);
  console.log(`args: ${args.join(', ')}`);
  const result = next({originalFunction, args});
  console.log(`result: ${result}`);
  return result;
};

const validationMiddleware = ({originalFunction, args, next}) => {
  if (args.some(arg => typeof arg != 'number')) throw new Error('not a number');
  return next({originalFunction, args});
};

calculatorWithMiddleware.addMiddleware(loggingMiddleware);
calculatorWithMiddleware.addMiddleware(validationMiddleware);

console.log(calculatorWithMiddleware.sum(3, 5));
console.log(calculatorWithMiddleware.storedValue);
console.log(calculatorWithMiddleware.product('a', 5));
```

The usage code initially defines the object `calculator`. This component consists of the functions `sum()` and `product()` as well as a stored value. Next, the class `MiddlewareProxy` is instantiated with the `calculator` as argument. Then, the function `loggingMiddleware()` is defined. Again, this operation consists of multiple steps. First, it logs the name of the original function together with the respective arguments. Then, it continues the execution of the chain. Afterwards, it logs the result and returns it back to the caller. The second middleware is the function `validationMiddleware()`, which verifies that all given arguments are numbers. Whenever it encounters another type, it throws an error. Both decorating operations are added to the proxy via the command `addMiddleware()`. Finally, multiple example invocations demonstrate the overall behavior.



Simplicity versus capabilities

The required signature of a middleware function for the component `MiddlewareProxy` is relatively complex. There are JavaScript libraries with a middleware mechanism that only require the callback to receive the argument `next`. While this simplifies the interface, it also limits the possibilities. For example, the parameter `originalFunction` can be used to completely reassign the actual function to execute.

Example: Chat

Consider extending and improving the implementation of a chat software. The underlying Domain Model contains two components. One part is the chat message, which consists of an identifier, an author reference, a timestamp and a text. All of its attributes are defined as immutable. Secondly, there is the chat itself, which encloses a constant identifier and a collection of messages. New entries can be appended and existing ones can be deleted. Both components require a conceptual identity and are therefore implemented as Entity type. With regard to persistence and consistency, they are combined into a single Aggregate with the chat as root Entity. Consequently, there is one single Repository component. For orchestrating and executing the actual uses cases, the software provides the according Application Services.

The first two examples show the implementations of the Entities for chat and chat message ([run code usage](#)):

Chat: Chat message Entity

```
class ChatMessage {  
    id; authorId; time; text;  
  
    constructor({id, authorId, time, text = ''}) {  
        Object.freeze(Object.assign(this, {id, authorId, time, text}));  
    }  
}
```

Chat: Chat Entity

```
class Chat {  
  
    id; #messages;  
  
    constructor({id, messages = []}) {  
        Object.defineProperty(this, 'id', {value: id, writable: false});  
        this.#messages = messages;  
    }  
  
    appendMessage(message) {  
        this.#messages.push(message);  
    }  
  
    deleteMessage(messageId) {  
        const index = this.#messages.findIndex(message => message.id === messageId);  
        if (index === -1) throw new Error('invalid message');  
        this.#messages.splice(index, 1);  
    }  
  
    get messages() { return this.#messages.slice(); }  
  
}
```

The third example provides the Repository component for the chat Aggregate:

Chat: Chat Repository

```
class ChatRepository extends ConcurrencySafeFilesystemRepository {  
  
    constructor({storageDirectory}) {  
        super({storageDirectory,  
            convertToEntity: data => new Chat({id: data.id,  
                messages: data.messages.map(messageData => new ChatMessage(messageData))),  
            convertToData: ({id, messages}) => ({id, messages}))};  
    }  
  
}
```

This is followed by the according Application Services component:

Chat: Application Services

```
class ApplicationServices {  
  
    #chatRepository;  
  
    constructor({chatRepository}) {  
        this.#chatRepository = chatRepository;  
    }  
  
    async createChat({chatId}) {  
        await this.#chatRepository.save(new Chat({id: chatId}));  
    }  
  
    async writeChatMessage({chatId, messageId, authorId, text}) {  
        const chat = await this.#chatRepository.load(chatId);  
        const time = Date.now();  
        const chatMessage = new ChatMessage({id: messageId, authorId, time, text});  
        chat.appendMessage(chatMessage);  
        await this.#chatRepository.save(chat);  
    }  
  
    async deleteChatMessage({chatId, messageId}) {  
        const chat = await this.#chatRepository.load(chatId);  
        chat.deleteMessage(messageId);  
        await this.#chatRepository.save(chat);  
    }  
  
    async getChatMessages({chatId}) {  
        const chat = await this.#chatRepository.load(chatId);  
        return chat.messages.map(message => {  
            const formattedTime = new Date(message.time).toLocaleString();  
            return `${message.authorId} (${formattedTime}): ${message.text}`;  
        }).join('\n');  
    }  
}
```

The class `ChatMessage` expresses the concept of an individual message. For simplicity reasons, all its fields are defined as immutable. The component `Chat` implements the chat concept. Its constructor defines attributes for an identifier and a collection of messages. Appending a new item is achieved with the command `appendMessage()`. Deleting an existing one is done via the function `deleteMessage()`. Access to the current list is provided through the getter `messages`.

The class `ChatRepository` extends the component `ConcurrencySafeFilesystemRepository` and provides the operations for correctly converting data and Entities. Finally, the class `ApplicationServices` implements the orchestration and execution of all use cases. Generating a human-readable log is done via the service `getChatMessages()`. This operation loads a chat, transforms its messages into domain-agnostic information and returns the result.

The next example demonstrates a basic usage of the chat software ([run code](#)):

Chat: Usage

```
const chatRepository = new ChatRepository({storageDirectory});
const applicationServices = new ApplicationServices({chatRepository});

const chatId = generateId(), messageId = generateId(), authorId = generateId();
await applicationServices.createChat({chatId});
await applicationServices.writeChatMessage({chatId, messageId, authorId,
  text: 'Hello!'});
console.log(await applicationServices.getChatMessages({chatId}));
await applicationServices.deleteChatMessage({chatId, messageId});
console.log(await applicationServices.getChatMessages({chatId}))
```

The code starts with creating an object of the `ChatRepository` component. Next, the class `ApplicationServices` is instantiated with the Repository as constructor argument. Afterwards, the identifiers for a chat, an exemplary message and an author are generated. This is followed by creating a chat Aggregate via the service `createChat()`. Then, a single message is written and appended through the execution of `writeChatMessage()`. Verifying the operational success is done with the help of the query `getChatMessages()`. The appended message is removed again by invoking `deleteChatMessage()`. Finally, the removal is verified with another call to `getChatMessages()`. This example illustrates the basic usage of the chat software. The implementation fulfills the described functional requirements. However, there are some scenarios that can potentially cause problems with regard to concurrency.

The next example illustrates a usage that produces concurrency conflicts ([run code](#)):

Chat: Usage with conflicts

```
const chatId = generateId();
const authorId1 = generateId(), messageId1 = generateId();
const authorId2 = generateId(), messageId2 = generateId();
await applicationServices.createChat({chatId});
await Promise.all([
    applicationServices.writeChatMessage({chatId, messageId: messageId1,
        authorId: authorId1, text: 'Hello!'}),
    applicationServices.writeChatMessage({chatId, messageId: messageId2,
        authorId: authorId2, text: 'Hello!'})
]);
console.log(await applicationServices.getChatMessages({chatId}));
```

Similar to the previous example, the code initially defines all required identifiers. In this case, two pairs of identities for an author and a message are generated. Then, a chat Aggregate is created by executing the command `createChat()`. This is followed by concurrently writing two messages. Afterwards, the current status is output using the query `getChatMessages()`. Executing the code produces a concurrency conflict during the attempt to write the messages. The crucial question is whether this circumstance is hinting to a design issue. For one, grouping messages together as common Aggregate ensures a definite order of items. At the same time, writing messages concurrently should be supported and must not produce a conflict. One possible solution for this is to implement an automatic retry behavior.

The next code provides a factory for creating a middleware to automatically retry failed service executions:

Chat: Create retry middleware

```
const createRetryMiddleware = ({retries}) =>
  async ({originalFunction, args, next}) => {
    while (retries-- > 0)
      try {
        return await next({originalFunction, args});
      } catch (error) {
        const isConcurrencyConflict = error.message === 'ConcurrencyConflict';
        if (!isConcurrencyConflict || retries <= 0) throw error;
        await timeout(15);
      }
  };
};
```

The last example shows a usage of the previously introduced middleware proxy and the retry middleware ([run code](#)):

Chat: Usage with retry

```
const applicationServices =
  new MiddlewareProxy(new ApplicationServices({chatRepository}));
applicationServices.addMiddleware(createRetryMiddleware({retries: 5}));

const chatId = generateId();
const authorId1 = generateId(), messageId1 = generateId();
const authorId2 = generateId(), messageId2 = generateId();
await applicationServices.createChat({chatId});
await Promise.all([
  applicationServices.writeChatMessage({chatId, messageId: messageId1,
    authorId: authorId1, text: 'Hello!'}),
  applicationServices.writeChatMessage({chatId, messageId: messageId2,
    authorId: authorId2, text: 'Hello!'})
]);
console.log(await applicationServices.getChatMessages({chatId}));
```

The function `createRetryMiddleware()` accepts a number of retries and returns a middleware to retry service executions upon encountering concurrency conflicts. Each function invocation is wrapped inside an asynchronous `while` loop. The default behavior is to execute the original and return its result. In case of a concurrency error, the middleware waits 15 milliseconds and issues the next attempt. Other errors are re-thrown to the outside. The usage code instantiates the class `MiddlewareProxy` and provides the previously created `ApplicationServices` object as target. Next, a retry middleware is created via the function `createRetryMiddleware()` and added to the proxy with the command `addMiddleware()`. The actual use case is identical to the previous example. However, due to the middleware, the concurrent writing of messages is retried until it succeeds.

Authentication and Authorization

Authentication and Authorization are security concepts for enabling access control. Simply put, Authentication is about identity and Authorization deals with permissions. For both concepts, there are various solution approaches. The most common form of authentication is the use of identifier and password. Typically, this is complemented with a session-based token mechanism. Authorization can be realized as Role-based Access Control (RBAC). Architecturally speaking, such implementations are infrastructural functionalities. Nevertheless, it can make sense to partially integrate security-related knowledge into a Domain Model. Both of the concepts are introduced in this chapter to exemplify how Application Services

can integrate security mechanisms. However, since the topics are broad and complex, they are not covered in full depth. Rather, this section gives a brief introduction and provides simplified example implementations.

Passwords and session tokens

The majority of consumer-facing software requires its users to provide an identifier and a password for an initial authentication. These values are compared to stored information, typically originating from a preceding registration. If the credentials match, the identity counts as confirmed. While this approach works well in stateful environments, it falls short when using a stateless communication protocol, such as HTTP. In such a case, both identifier and password would need to be included in every request. However, sending credentials repeatedly to another party is a security risk. This issue can be mitigated by generating a token that acts as authentication surrogate. After an initial login, only the token is passed along with every request. The main advantage is its temporary validity and its replaceability.

The following code provides a registry component to maintain authentication tokens ([run code usage](#)):

Security: Authentication token registry

```
class AuthenticationTokenRegistry {  
  
    #storageDirectory;  
  
    constructor({storageDirectory}) {  
        this.#storageDirectory = storageDirectory;  
        mkdirSync(this.#storageDirectory, {recursive: true});  
    }  
  
    async assignToken(subjectId, token) {  
        await writeFile(this.#getFilePath(subjectId, token), '');  
    }  
  
    isTokenValid(subjectId, token) {  
        return access(this.#getFilePath(subjectId, token))  
            .then(() => true).catch(() => false);  
    }  
  
    async removeToken(subjectId, token) {  
        await unlink(this.#getFilePath(subjectId, token));  
    }  
}
```

```
#getFilePath(subjectId, token) {
    return `${this.#storageDirectory}/${subjectId}.${token}`;
}

}
```

The class `AuthenticationTokenRegistry` provides a persistent registry for authentication tokens associated with arbitrary subjects. Apart from real users interacting with a software, subsystems commonly need to communicate with each other. For describing both human and non-human clients, the generic term “subject” fits best. The command `assignToken()` creates an empty file, for which the filename is a combination of subject identifier and token. Testing a token for its validity is achieved with the query `isValid()`. This function verifies the existence of the according file. The operation `removeToken()` is responsible for invalidating a token through file deletion. All operations are asynchronous, as they work with the filesystem. While the registry does not dictate a specific token format, it is recommended to use unique values such as UUIDs.

Path-based access control

Almost every software that applies the concept of authentication also deals with some form of authorization. While identifying individual subjects is an important matter, it is normally a means to an end. The provided information is typically used as foundation for determining whether access is existent. One simple implementation for authorization is a path-based access control, where the term “path” stands for a hierarchical unique identifier. Every such value can be associated with any number of authentication subjects. Depending on the use case, the paths can represent resources, actions or even roles. Also, the values may contain wildcards to express permissions in a generalized way. For example, while the path `user/john-doe` stands for the access to one user, the value `user/*` represents all of them.

The next example shows a registry component for access paths ([run code usage](#)):

Security: Access registry

```
class AccessRegistry {

    #storageDirectory;

    constructor({storageDirectory}) {
        this.#storageDirectory = storageDirectory;
    }

    async grantAccess(subjectId, accessPath) {
        const accessDirectory = this.#convertToDirectory(accessPath);
        await mkdir(accessDirectory, {recursive: true});
        await writeFile(`${accessDirectory}/${subjectId}`, '');
    }

    async verifyAccess(subjectId, accessPath) {
        const fullAccessDirectory = this.#convertToDirectory('full-access');
        await access(`${fullAccessDirectory}/${subjectId}`)
            .catch(() => access(`${this.#convertToDirectory(accessPath)}/${subjectId}`))
            .catch(() => { throw new Error('access denied'); });
    }

    async revokeAccess(subjectId, accessPath) {
        const accessDirectory = this.#convertToDirectory(accessPath);
        await unlink(`${accessDirectory}/${subjectId}`);
    }

    #convertToDirectory(accessPath) {
        this.#verifyAccessPath(accessPath);
        return `${this.#storageDirectory}/${accessPath}`;
    }

    #verifyAccessPath(accessPath) {
        if (/[^a-zA-Z0-9/-]/gi.test(accessPath)) throw new Error('invalid path');
    }
}
```

The class `AccessRegistry` implements a persistent registry for access paths associated with authentication subjects. Adding an identifier to a path is done with the command `grantAccess()`. The operation first converts the provided path into a directory name. Disallowed characters cause an exception to be thrown. Next, the function ensures the

existence of the target directory. Finally, an empty file is created, with the subject identifier as filename. Testing whether access is granted is achieved with the operation `verifyAccess()`. The functionality is implemented as command that throws an exception if an identifier is not associated with a path. Compared to a query approach, this variant is more intuitive to use. The function `revokeAccess()` removes a subject identifier from an access path. Again, all operations are asynchronous.

Example: File sharing

Consider implementing the Domain Model, the Application Services, the authentication and the authorization for a file sharing software. The goal is for users to be able to upload and download files and share them with others. With regard to security, it is crucial to prevent unauthorized access. The Domain Model consists of two components. For one, there is the user part, which encloses identifier, username and password. Secondly, there is the file component, which contains identifier, owner identity, name, content and additional users with access. Apart from the users to share files with, all attributes of both Domain Model parts are considered immutable. The required functionalities are to create new users, perform authentication, upload and download files, and share them with others.

The first examples show the implementations of the Entities for users and files ([run code usage](#)):

File sharing: User Entity

```
const User = function({id, username, password}) {
  Object.freeze(Object.assign(this, {id, username, password}));
};
```

File sharing: File Entity

```
class File {

  id; ownerId; name; content; #usersWithAccess = [];

  constructor({id, ownerId, name, content}) {
    Object.assign(this, {id, ownerId, name, content});
  }

  addUserAccess(userId) {
    this.#usersWithAccess.push(userId);
  }
}
```

```
get usersWithAccess() { return this.#usersWithAccess.slice(); }

}
```

The next code provides a factory function for creating an authentication middleware ([run code usage](#)):

File sharing: Authentication middleware factory

```
const createAuthenticationMiddleware = (
  {authenticationTokenRegistry, authenticationExtractor} =>
  async ({originalFunction, args, next}) => {
    if (originalFunction.bypassAuthentication)
      return next({originalFunction, args});
    const {subjectId, token} = authenticationExtractor(...args) || {};
    if (!subjectId || !token) throw authenticationError({originalFunction, args});
    const isAuthenticated =
      await authenticationTokenRegistry.isTokenValid(subjectId, token);
    if (!isAuthenticated) throw authenticationError({originalFunction, args});
    return next({originalFunction, args});
};

const authenticationError = ({originalFunction, args}) => new Error(
  `authentication failed for ${originalFunction.name}(${JSON.stringify(args)})`);
```

The factory `createAuthenticationMiddleware()` creates an authentication function that can be consumed by a middleware proxy. As arguments, it expects a token registry and an extractor function. The extractor argument determines how to retrieve authentication information. As return value, the factory yields an operation that consists of multiple steps. First, it checks whether the target function has the attribute `bypassAuthentication`, in which case it is executed immediately. Next, it invokes the extractor operation to retrieve the authentication metadata. If the information is missing, an exception is thrown. In case of success, the subject identifier and token are passed to the registry query `isTokenValid()`. Invalid tokens also trigger an exception. As last step, the original target function is executed. The function `authenticationError()` creates a descriptive error instance.

The following component provides the file sharing Application Services:

File sharing: Application Services

```
class ApplicationServices {  
  
    #userRepository; #fileRepository; #authenticationTokenRegistry; #accessRegistry;  
  
    constructor({userRepository, fileRepository, authenticationTokenRegistry, accessRegistry}) {  
        this.#userRepository = userRepository;  
        this.#fileRepository = fileRepository;  
        this.#authenticationTokenRegistry = authenticationTokenRegistry;  
        this.#accessRegistry = accessRegistry;  
        this.createAndLoginUser.bypassAuthentication = true;  
    }  
  
    async createAndLoginUser({userId, username, password, authenticationToken}) {  
        await this.#userRepository.save(new User({id: userId, username, password}));  
        await this.#authenticationTokenRegistry.assignToken(userId, authenticationToken);  
    }  
  
    async uploadFile(data, {authentication: {subjectId}}) {  
        const {fileId: id, ownerId, name, content} = data;  
        if (subjectId !== ownerId) throw new Error('access denied');  
        await this.#fileRepository.save(new File({id, ownerId, name, content}));  
        await this.#accessRegistry.grantAccess(subjectId, `file/${id}`);  
    }  
  
    async downloadFile({fileId}, {authentication: {subjectId}}) {  
        await this.#accessRegistry.verifyAccess(subjectId, `file/${fileId}`);  
        const {name, content} = await this.#fileRepository.load(fileId);  
        return {name, content};  
    }  
  
    async addUserAccessToFile({fileId, userId}, {authentication: {subjectId}}) {  
        const file = await this.#fileRepository.load(fileId);  
        if (subjectId !== file.ownerId) throw new Error('access denied');  
        file.addUserAccess(userId);  
        await this.#fileRepository.save(file);  
        await this.#accessRegistry.grantAccess(userId, `file/${fileId}`);  
    }  
}
```

The class `ApplicationServices` implements all the described use cases for the file sharing

software. Its constructor requires Repositories for both files and users, an authentication token registry and an access registry. For brevity reasons, the creation of a user and its login are combined into a single use case. Overall, the component demonstrates different levels of Authentication and Authorization. Creating a new user can be done anonymously. Uploading a file as well as granting access to others is only allowed for the owner of a document. For both of these actions, the authorization consists of comparing authentication information to Domain Model knowledge. The third scenario is downloading a file. In this case, the functionality is secured through checking the access registry for an according path.

The service `createAndLoginUser()` is marked to bypass the authentication via the according attribute. This operation both creates a user and assigns the provided authentication token. The function `uploadFile()` is responsible for creating a file and granting access to its owner. As authorization, it checks whether the requesting user is the same as the target owner. The command `downloadFile()` returns the name and the content of an individual file. For this operation, the authorization part checks whether the requesting user has access to the according path. Finally, the function `addUserAccessToFile()` both adds a user identifier to the according Entity and grants access via the registry. Similar to uploading a file, the authorization-specific code checks whether the requesting subject matches the file owner.

The final example demonstrates the usage of all components ([run code](#)):

File sharing: Usage code

```
const userRepository = new FilesystemRepository({
  storageDirectory: `${rootDirectory}/user`,
  convertToData: entity => entity, convertToEntity: data => new User(data));
const fileRepository = new FilesystemRepository({
  storageDirectory: `${rootDirectory}/file`,
  convertToData: entity => entity, convertToEntity: data => new File(data));

const authenticationTokenRegistry = new AuthenticationTokenRegistry(
  {storageDirectory: `${rootDirectory}/authentication`});
const accessRegistry = new AccessRegistry(
  {storageDirectory: `${rootDirectory}/authorization`});

const applicationServices = new MiddlewareProxy(new ApplicationServices({
  userRepository, fileRepository, accessRegistry, authenticationTokenRegistry)));
const authenticationExtractor = (_, metadata = {}) => metadata.authentication;
const middleware = createAuthenticationMiddleware(
  {authenticationTokenRegistry, authenticationExtractor});
applicationServices.addMiddleware(middleware);

const user1Id = generateId(), user2Id = generateId(), fileId = generateId();
```

```
const user1Authentication = {subjectId: user1Id, token: generateId()};
const user2Authentication = {subjectId: user2Id, token: generateId()};

const tryDownloadFile = async ({fileId, authentication = {}}) => {
  const message = `download ${fileId} for subject ${authentication.subjectId}`;
  try {
    await applicationServices.downloadFile({fileId}, {authentication});
    console.log(`#${message} success`);
  } catch (error) { console.log(message + error.message); }
};

await applicationServices.createAndLoginUser({userId: user1Id, username: 'johndoe',
  password: 'secret1', authenticationToken: user1Authentication.token});
await applicationServices.uploadFile(
  {fileId, name: 'notes.txt', ownerId: user1Id, content: 'This and that.'},
  {authentication: user1Authentication});
await tryDownloadFile({fileId});
await tryDownloadFile({fileId, authentication: user1Authentication});
await applicationServices.createAndLoginUser({userId: user2Id, username: 'janedoe',
  password: 'secret2', authenticationToken: user2Authentication.token});
await tryDownloadFile({fileId, authentication: user2Authentication});
console.log(`granting user ${user2Id} access to file`);
await applicationServices.addUserAccessToFile(
  {fileId, userId: user2Id}, {authentication: user1Authentication});
await tryDownloadFile({fileId, authentication: user2Authentication});
```

The code starts with creating one Repository for users and another one for file Entities. Next, both the classes AuthenticationTokenRegistry and AccessRegistry are instantiated. Afterwards, an instance of the component ApplicationServices is created and decorated with a middleware proxy. This is followed by defining an extractor operation that retrieves authentication information from a metadata argument. As next step, an authentication middleware function is created and added to the Application Services instance. As preparation for the actual usage example, the identifiers and the metadata for two users and one file are defined. Also, the helper function `tryDownloadFile()` is implemented. This operation attempts to download a file and outputs an informational message to the console.

The actual usage starts with executing the service `createAndLoginUser()` to create a first user. Afterwards, a file is added by calling the operation `uploadFile()` and passing in sample data. Next, this file is attempted to be downloaded, both anonymously and authenticated as the owner. This is followed by the creation of a second user and a download attempt using its authentication information. Only afterwards, access to the file is granted. Then, the newly

created user initiates another download attempt. Executing the code demonstrates different aspects. For one, anonymous downloads always fail due to the missing authentication. Secondly, the owner of a file is always allowed to access it. And finally, other users cannot download files, unless access is explicitly granted.



Multiple transactions per Application Service

As explained earlier, an Application Service should ideally only affect a single transaction. The above examples violate this recommendation. Many of the services affect both one Aggregate transaction and another one related to security. While this is not automatically an issue, it can be avoided. One approach is to use Domain Events and update security information in a deferred way.

Sample Application: Application Services

This section illustrates the implementation of Application Services and basic variants of authentication and authorization for the Sample Application. Consequently, the focus lies on other architectural layers than the Domain part. Only the Domain Model implementation of the project context requires a minor refactoring due to considerations on transactions and consistency. For each of the three context implementations, the respective Application Services and security aspects are described and illustrated separately. Wherever possible, the service operations adhere to the recommended naming convention. Note that this section does not contain the full source code of the according services. For brevity reasons, only relevant parts are shown and other functionalities that follow the same patterns are left out.



Exemplary usage of security mechanisms

The usage of authentication and authorization is limited to this chapter. This means, the following chapters of the book exclude them again. The reason is that both aspects introduce additional complexity and can be counterproductive to the illustration of the main concepts.

Infrastructure code

As with previous chapters, this section introduces additional generic functionalities to the Sample Application. For enabling middleware in Application Services, the `MiddlewareProxy`

class is added. The components `AuthenticationTokenRegistry`, `AccessRegistry` and the authentication middleware factory are required for security purposes. Furthermore, the access registry is extended with three operations. The command `grantFullAccess()` adds a subject identifier to the path `full-access`, which is first checked upon when executing the function `verifyAccess()`. This allows to create administrative users as defined in the Domain Model. The command `grantImplicitAccess()` creates an alias for a path, which allows to grant derived access. For example, access to a task can be granted implicitly when it is added to a task board. The operation `revokeImplicitAccess()` is responsible for deleting an access path alias.

The following code shows the additional functions for the access registry component:

Infrastructure code: Additional access registry functions

```
grantFullAccess(subjectId) {
    return this.grantAccess(subjectId, 'full-access');
}

async grantImplicitAccess(originalAccessPath, implicitAccessPath) {
    const originalDirectory = this.#convertToDirectory(originalAccessPath);
    const implicitDirectory = this.#convertToDirectory(implicitAccessPath);
    const implicitDirectoryParent = path.join(implicitDirectory, '../');
    await Promise.all([mkdir(originalDirectory, {recursive: true}),
        mkdir(implicitDirectoryParent, {recursive: true})]);
    await symlink(path.resolve(originalDirectory), implicitDirectory);
}

async revokeImplicitAccess(implicitAccessPath) {
    const symlinkDirectory = this.#convertToDirectory(implicitAccessPath);
    await rmdir(symlinkDirectory);
}
```



Admin access paths

Executing the command `grantFullAccess()` makes the operation `verifyAccess()` always succeed for a given subject identifier. This is even true for paths to which no subject has explicitly access to. This behavior enables to use separate values for different admin actions, such as “admin/create-user”. The approach can be beneficial when aiming to introduce the values as real paths in the future.

User context

Besides its domain-specific purpose, the user context implementation also contains the foundation for authentication through user creation and identity verification. Although access control is mostly an infrastructural topic, a Domain Model is typically required to provide a mechanism for validating credentials. The user context code provides two functionalities for this purpose. One is the Repository query `findUserByEmail()`, which enables to identify a user via its e-mail address. Secondly, there is the Entity function `isPasswordMatching()`, which determines the validity of a password. These operations can be used for the initial authentication process. In case of valid credentials, an artificial token is assigned and used for subsequent identity verification. Note that this token only exists in the Infrastructure Layer and is not part of the Domain Model.

The following code shows the most relevant parts of the Application Services class for the user context:

User context: Application Services

```
class UserApplicationServices {

    #userRepository; #userFactory; #emailRegistry; #hashPassword;
    #authenticationTokenRegistry; #accessRegistry; #verifyAccess;

    constructor({userRepository, userFactory, emailRegistry, hashPassword,
        authenticationTokenRegistry, accessRegistry}) {
        /* .. assignment of constructor arguments to private fields .. */
        this.#verifyAccess = accessRegistry.verifyAccess.bind(accessRegistry);
        this.loginUser.bypassAuthentication = true;
        this.initializeEmailRegistry.bypassAuthentication = true;
    }

    async createUser({userId, username, emailAddress, password, role}, metadata) {
        const subjectId = metadata.authentication.subjectId;
        await this.#verifyAccess(subjectId, 'admin/create-user');
        const user = this.#userFactory.createUser({id: userId, username,
            password: this.#hashPassword(password), role: new Role(role),
            emailAddress: new EmailAddress(emailAddress)});
        await this.#userRepository.save(user);
        await user.role.equals(new Role('admin')) ?
            this.#accessRegistry.grantFullAccess(userId) :
            this.#accessRegistry.grantAccess(userId, `user/${userId}`));
    }
}
```

```

async loginUser({emailAddress, password, authenticationToken}) {
    const user =
        await this.#userRepository.findUserByEmail(new EmailAddress(emailAddress));
    if (user.password != this.#hashPassword(password))
        throw new Error('invalid password');
    await this.#authenticationTokenRegistry.assignToken(
        user.id, authenticationToken);
}

async updateUserEmailAddress({userId, emailAddress}, metadata) {
    const subjectId = metadata.authentication.subjectId;
    await this.#verifyAccess(subjectId, `user/${userId}`);
    const user = await this.#userRepository.load(userId);
    user.emailAddress = new EmailAddress(emailAddress);
    await this.#userRepository.save(user);
}

/* .. updateUserUsername(), updateUserPassword(), updateUserRole() .. */

async initializeEmailRegistry() {
    (await this.#userRepository.loadAll()).forEach(user =>
        this.#emailRegistry.setUserEmailAddress(user.id, user.emailAddress));
}

```

The next example provides a Domain Event handlers component for synchronizing the e-mail registry state:

User context: Domain Event handlers

```

class UserDomainEventHandlers {

    constructor({emailRegistry, eventBus}) {
        this.activate = () => {
            eventBus.subscribe(UserEmailAddressAssignedEvent.type, ({data}) => {
                emailRegistry.setUserEmailAddress(
                    data.userId, new EmailAddress(data.emailAddress));
            });
        };
    }

}

```

The class `UserApplicationServices` implements all identity-related use cases together with a basic authentication. Creating and authorizing a new user is done with the command `createUser()`. While normal users can initially only access their own data, admins get full access. Note that executing this operation requires authorization itself. This aspect is illustrated and solved exemplarily by the following usage code. The command `loginUser()` verifies an identity and assigns an authentication token in case of success. Due to the authentication bypass flag, the operation can be executed anonymously. The service `updateUserEmailAddress()` is one of multiple functionalities to modify user Entities. Populating the e-mail registry is done with the command `initializeEmailRegistry()`. The class `UserDomainEventHandlers` implements a subscriber function to update the registry state in response to “`UserEmailAddressAssigned`” events.

The next example shows a simplified usage of the Application Services component together with security ([run code](#)):

User context: Overall usage

```
const userApplicationServices = new MiddlewareProxy(
  new UserApplicationServices({userRepository, userFactory, emailRegistry,
    hashPassword: createMd5Hash, authenticationTokenRegistry, accessRegistry}));
const authenticationExtractor = (_, metadata = {}) => metadata.authentication;
userApplicationServices.addMiddleware(createAuthenticationMiddleware(
  {authenticationTokenRegistry, authenticationExtractor}));
const userDomainEventHandlers = new UserDomainEventHandlers(
  {emailRegistry, eventBus});
userDomainEventHandlers.activate();

const random = `${Date.now()}`;
const userId = generateId(), adminId = generateId(), systemUserId = generateId();
const userEmail = `jd${random}@example.com`;
const adminEmail = `admin${random}@example.com`;
const authenticationMetadata = {
  system: {subjectId: systemUserId, token: generateId()},
  admin: {subjectId: adminId, token: generateId()},
};

await authenticationTokenRegistry.assignToken(
  authenticationMetadata.system.subjectId, authenticationMetadata.system.token);
await accessRegistry.grantFullAccess(systemUserId);

await userApplicationServices.initializeEmailRegistry();
```

```
await userApplicationServices.createUser({userId: adminId, username: 'admin',
  role: 'admin', password: 'pw1', emailAddress: adminEmail},
  {authentication: authenticationMetadata.system});
await userApplicationServices.loginUser({emailAddress: adminEmail,
  password: 'pw1', authenticationToken: authenticationMetadata.admin.token});
await userApplicationServices.createUser({userId: userId, username: 'johndoe',
  role: 'user', password: 'pw2', emailAddress: userEmail},
  {authentication: authenticationMetadata.admin});
await userApplicationServices.updateUserEmailAddress({userId,
  emailAddress: `john-doe${random}@example.com`},
  {authentication: authenticationMetadata.admin});
const {id, username, emailAddress} = await userRepository.load(userId);
console.log({id, username, emailAddress});
```

The usage code illustrates how to create an initial admin user. First, a system-owned subject identifier and an authentication token are created. Then, these values are passed to both the token registry and the access registry. Afterwards, the service function `createUser()` can be executed with the system authentication information passed in as metadata.

Project context

The project context incorporates two use cases that affect multiple consistency boundaries. For one, adding a new team member includes creating a member and updating the target team. For this part, it is acceptable to combine both transactions in a single service. In the worst case, creating a member without adding it to a team produces an orphaned Aggregate. The second process is the introduction of a new project, which includes the creation of a team and a task board. Unlike the local team component, the task board is part of another conceptual boundary. In this case, it makes most sense to connect the individual steps via event notifications. Consequently, the project Entity implementation must yield a Domain Event whenever a new instance is created.

The first example introduces a Domain Event that represents the creation of a new project:

Project context: Domain Event

```
const ProjectCreatedEvent = createEventType('ProjectCreated',
  {projectId: 'string', name: 'string', teamId: 'string', taskBoardId: 'string'});
```

The second code shows a reworked version of the constructor for the project Entity:

Project context: Project constructor

```
constructor({id, name, ownerId, teamId, taskBoardId, isExistingProject = false}) {
    verify(id && name && ownerId && teamId && taskBoardId, 'valid data');
    Object.defineProperties(this, {
        id: {value: id, writable: false},
        ownerId: {value: ownerId, writable: false},
        teamId: {value: teamId, writable: false},
        taskBoardId: {value: taskBoardId, writable: false},
    });
    this.name = name;
    if (!isExistingProject) this.#newDomainEvents.push(
        new ProjectCreatedEvent({projectId: id, name, teamId, taskBoardId}));
}
```

The next example introduces a factory for creating and reconstituting project Entities:

Project context: Project factory

```
class ProjectFactory {

    createProject({id, name, ownerId, teamId, taskBoardId}) {
        return new Project(
            {id, name, ownerId, teamId, taskBoardId, isExistingProject: false});
    }

    reconstituteProject({id, name, ownerId, teamId, taskBoardId}) {
        return new Project(
            {id, name, ownerId, teamId, taskBoardId, isExistingProject: true});
    }
}
```

This is followed by an updated constructor of the project Repository component:

Project context: Project Repository constructor

```
constructor({storageDirectory, projectFactory}) {
    super({storageDirectory,
        convertToData: ({id, name, ownerId, teamId, taskBoardId}) =>
            ({id, name, ownerId, teamId, taskBoardId}),
        convertToEntity: data => projectFactory.reconstituteProject(data)});
}
```

The Domain Event type `ProjectCreatedEvent` represents the creation of a new project. Its data structure consists of an identifier, a name and identities for a team and a task board. The project context itself must process this event type and create a new team for each occurrence. Similarly, the task board context is responsible for creating a new task board as a reaction. The reworked constructor of the `Project` component expects the additional argument `isExistingProject` and defines a collection for Domain Events. In case of a new instance, an event of the type “`ProjectCreated`” is instantiated and added to the collection. The class `ProjectFactory` encapsulates the Entity creation. Finally, the updated constructor of the component `ProjectRepository` re-uses the factory for the operation `convertToEntity()`.

Overall, the project context contains use cases for projects, teams and individual members. Other than the user part, the Domain Model consists of multiple separate Aggregate types. One is for team members, a second one is for teams and the third is for projects. Therefore, the design question is how to structure the according Application Services implementation. One approach is to introduce a separate class per Aggregate type. This aims to create a clear mapping between consistency boundaries and service functionalities. Another option is to gather all services in a single component. This approach helps to avoid an artificial separation of logically related use cases. Due to the overall low complexity of this context, the second approach is likely to be more useful.

The following code provides the relevant parts of the Application Services for the project context:

Project context: Application Services

```
class ProjectApplicationServices {  
  
    #projectRepository; #teamRepository; #teamMemberRepository;  
    #projectFactory; #accessRegistry; #verifyAccess;  
  
    constructor({projectRepository, teamRepository, teamMemberRepository,  
        projectFactory, accessRegistry}) {  
        /* .. assignment of constructor arguments to private fields .. */  
        this.#verifyAccess = accessRegistry.verifyAccess.bind(accessRegistry);  
    }  
  
    async createProject({projectId, name, ownerId, teamId, taskBoardId},  
        {authentication: {subjectId}}) {  
        await this.#verifyAccess(subjectId, 'admin/create-project');  
        await this.#projectRepository.save(this.#projectFactory.createProject(  
            {id: projectId, name, ownerId, teamId, taskBoardId}));  
    }  
  
    async addTeamMemberToTeam({teamId, teamMemberId, userId, role}, metadata) {  
        const {authentication: {subjectId}} = metadata;  
        await this.#verifyAccess(subjectId, `team/${teamId}`);  
        await this.#teamMemberRepository.save(new TeamMember(  
            {id: teamMemberId, userId, role: new Role(role)}));  
        const team = await this.#teamRepository.load(teamId);  
        team.addMember(teamMemberId);  
        await this.#teamRepository.save(team);  
        await this.#accessRegistry.grantAccess(userId, `team/${team.id}`);  
    }  
  
    async findProjectsByCollaboratingUser({userId}) {  
        const members = await this.#teamMemberRepository.findTeamMembersByUser(userId);  
        const projects = await Promise.all(members.map(async teamMemberId => {  
            const team = await this.#teamRepository.findTeamByTeamMember(teamMemberId);  
            return this.#projectRepository.findProjectByTeam(team.id);  
        }));  
        return projects.map(({id, name, ownerId, teamId, taskBoardId}) =>  
            ({id, name, ownerId, teamId, taskBoardId}));  
    }  
  
    async findProjectsByOwner({userId}) {  
        const projects = this.#projectsRepository.findProjectsByOwner(userId);  
        return projects.map(({id, name, ownerId, teamId, taskBoardId}) =>
```

```
        ({id, name, ownerId, teamId, taskBoardId}));  
    }  
  
    /* .. removeTeamMemberFromTeam(), updateTeamMemberRole() .. */  
  
}
```

The next example provides a Domain Event Handlers component that creates a new team for every new project:

Project context: Domain Event handlers

```
class ProjectDomainEventHandlers {  
  
    constructor({teamRepository, accessRegistry, eventBus}) {  
        this.activate = () => {  
            eventBus.subscribe(ProjectCreatedEvent.type, async ({data}) => {  
                await teamRepository.save(new Team({id: data.teamId}));  
                await accessRegistry.grantImplicitAccess(  
                    `team/${data.teamId}`, `project/${data.projectId}`);  
            });  
        };  
    };  
  
    }  
}
```

The class `ProjectApplicationServices` implements all project-specific use cases. Creating a project is done via the command `createProject()`, which can only be executed by admin users. The service persists a new project Aggregate and causes to publish a “ProjectCreated” event. Executing the operation `addTeamMemberToTeam()` introduces a new team member and adds it to a team. Determining all associated projects for a given user is done with the queries `findProjectsByCollaboratingUser()` and `findProjectsByOwner()`. The component `ProjectDomainEventHandlers` is responsible for creating a team for every new project. Invoking its command `activate()` registers a subscriber function for “ProjectCreated” events. For one, this operation creates and persists a new team Aggregate. Secondly, it executes the command `grantImplicitAccess()` to grant project access for all future members of the new team.

The following code provides an exemplary usage of the project Application Services ([run code](#)):

Project context: Overall usage

```
const projectApplicationServices = new MiddlewareProxy(
  new ProjectApplicationServices({teamMemberRepository,
    teamRepository, projectRepository, projectFactory, accessRegistry}));
projectApplicationServices.addMiddleware(authenticationMiddleware);
const projectDomainEventHandlers = new ProjectDomainEventHandlers(
  {teamRepository, accessRegistry, eventBus});
projectDomainEventHandlers.activate();

const teamId = generateId(), teamMemberId = generateId();
const projectId = generateId(), taskBoardId = generateId();
const userId = generateId(), adminId = generateId();
const authenticationMetadata = {user: {subjectId: userId, token: generateId()},
  admin: {subjectId: adminId, token: generateId()}};

await authenticationTokenRegistry.assignToken(
  authenticationMetadata.user.subjectId, authenticationMetadata.user.token);
await authenticationTokenRegistry.assignToken(
  authenticationMetadata.admin.subjectId, authenticationMetadata.admin.token);
await accessRegistry.grantFullAccess(adminId);

await projectApplicationServices.createProject(
  {projectId, name: 'Test Project', ownerId: adminId, teamId, taskBoardId},
  {authentication: authenticationMetadata.admin});
await timeout(125);
await projectApplicationServices.addTeamMemberToTeam({teamId, teamMemberId,
  userId, role: 'developer'}, {authentication: authenticationMetadata.admin});
await projectApplicationServices.updateProjectName(
  {projectId, name: 'Test Project v2'},
  {authentication: authenticationMetadata.user});
console.log(await projectApplicationServices.findProjectsByCollaboratingUser(
  {userId}, {authentication: authenticationMetadata.user}));
```

Task board context

The task board context encloses all aspects related to tasks and task boards. Its Domain Model implementation contains two Aggregate types, one for each of the separate concepts. There is one process within the context that affects both consistency boundaries. Creating a task and adding it to a board should be an atomic action. Similar to the addition of team members, a pragmatic solution approach is to incorporate both transactions within

a single service. The worst case scenario is that a task is created, but never added to a board. While this circumstance is undesirable, it does not risk producing a corrupted state. The Application Services for the task board context are implemented as single class due to the overall manageable complexity.



Services that affect multiple transactions

Both the project and the task board context violate the recommendation of only affecting a single transaction per service. This is only feasible because an interruption of the corresponding processes does not risk a corruption of state. Still, the implementation produces service operations that are not as deterministic and transparent as when only affecting single transactions.

The first example shows the most important use case operations of the Application Services for the task board context:

Task board context: Application Services

```
class TaskBoardApplicationServices {

    #taskRepository; #taskBoardRepository; #accessRegistry; #verifyAccess;

    constructor({taskRepository, taskBoardRepository, accessRegistry}) {
        /* .. */
    }

    async addNewTaskToTaskBoard(data, metadata) {
        const {taskId, taskBoardId, title, description, status, assigneeId} = data;
        const {authentication: {subjectId}} = metadata;
        await this.#verifyAccess(subjectId, `task-board/${taskBoardId}`);
        const task = new Task({id: taskId, title, description, status, assigneeId});
        await this.#taskRepository.save(task);
        const taskBoard = await this.#taskBoardRepository.load(taskBoardId);
        taskBoard.addTask(TaskSummary.createFromTask(task));
        await this.#taskBoardRepository.save(taskBoard);
        await this.#accessRegistry.grantImplicitAccess(
            `task-board/${taskBoardId}`, `task/${taskId}`);
    }

    async updateTaskTitle({taskId, title}, {authentication: {subjectId}}) {
        await this.#verifyAccess(subjectId, `task/${taskId}`);
        const task = await this.#taskRepository.load(taskId);
        task.title = title;
    }
}
```

```

        await this.#taskRepository.save(task);
    }

async removeTaskFromTaskBoard({taskBoardId, taskId}, metadata) {
    const {subjectId} = metadata.authentication;
    await this.#verifyAccess(subjectId, `task-board/${taskBoardId}`);
    const taskBoard = await this.#taskBoardRepository.load(taskBoardId);
    taskBoard.removeTask(taskId);
    await this.#taskBoardRepository.save(taskBoard);
    await this.#accessRegistry.revokeImplicitAccess(`task/${taskId}`);
}

async findTaskBoardTasks({taskBoardId, status = ''}, {authentication}) {
    await this.#verifyAccess(authentication.subjectId, `task-board/${taskBoardId}`);
    const taskBoard = await this.#taskBoardRepository.load(taskBoardId);
    return taskBoard.getTasks(status);
}

/* ... updateTaskDescription(), updateTaskAssignee(), updateTaskStatus() ... */

}

```

The next code implements a component for all necessary Domain Event handlers:

Task board context: Domain Event handlers

```

class TaskBoardDomainEventHandlers {

constructor({taskRepository, taskBoardRepository, accessRegistry, eventBus}) {

    this.activate = () => {
        eventBus.subscribe('ProjectCreated', async ({data}) => {
            await taskBoardRepository.save(new TaskBoard({id: data.taskBoardId}));
            await accessRegistry.grantImplicitAccess(
                `team/${data.teamId}`, `task-board/${data.taskBoardId}`);
        });
        eventBus.subscribe(TaskStatusChangedEvent.type, async ({data}) => {
            const taskBoard =
                await taskBoardRepository.findTaskBoardByTaskId(data.taskId);
            if (!taskBoard) return;
            taskBoard.updateTaskStatus(data.taskId, data.status);
            await taskBoardRepository.save(taskBoard);
        });
        eventBus.subscribe('TeamMemberRemovedFromTeam', async event => {

```

```
        const {data: {teamMemberId}} = event;
        const tasks = await taskRepository.findTasksByAssigneeId(teamMemberId);
        await Promise.all(tasks.map(async task => {
            if (task.status === 'in progress') task.status = 'todo';
            task.assigneeId = undefined;
            await taskRepository.save(task);
        }));
    });
};

}

}
```

The class `TaskBoardApplicationServices` implements all use cases for the task board context. Creating a new task and adding it to a board is done via the command `addNewTaskToTaskBoard()`. The operation first verifies the required access. In case of success, it persists a new task and attaches an according summary to the target board. Also, the service grants implicit access to all subjects with board access. The operation `updateTaskTitle()` is one of multiple services for task modifications. The command `removeTaskFromTaskBoard()` is an example for using the operation `revokeImplicitAccess()`. Domain Event notifications are processed by the component `TaskBoardDomainEventHandlers`. For every new project, a task board is created and persisted. Team member removals cause to un-assign affected tasks. Finally, every task status change triggers a task summary update.

The final code example shows an exemplary usage of the Application Services for the task board context ([run code](#)):

Task board context: Overall usage

```
const taskBoardApplicationServices =
    new MiddlewareProxy(new TaskBoardApplicationServices(
        {taskRepository, taskBoardRepository, accessRegistry}));
const authenticationExtractor = (_, metadata = {}) => metadata.authentication;
const authenticationMiddleware = createAuthenticationMiddleware(
    {authenticationTokenRegistry, authenticationExtractor});
taskBoardApplicationServices.addMiddleware(authenticationMiddleware);
const taskBoardDomainEventHandlers = new TaskBoardDomainEventHandlers(
    {taskRepository, taskBoardRepository, accessRegistry, eventBus});
taskBoardDomainEventHandlers.activate();
```

```
const adminUserId = generateId();
const authenticationMetadata = {subjectId: adminUserId, token: generateId()};
const taskBoardId = generateId(), taskId = generateId();

await authenticationTokenRegistry.assignToken(
  authenticationMetadata.subjectId, authenticationMetadata.token);
await accessRegistry.grantFullAccess(adminUserId);

eventBus.publish({type: 'ProjectCreated', data: {taskBoardId}});
await timeout(125);
await taskBoardApplicationServices.addNewTaskToTaskBoard({taskId, taskBoardId,
  title: 'write tests', description: 'write unit tests for new feature',
  {authentication: authenticationMetadata}});
await taskBoardApplicationServices.updateTaskAssignee(
  {taskId, assigneeId: generateId()}, {authentication: authenticationMetadata});
await taskBoardApplicationServices.updateTaskStatus(
  {taskId, status: 'in progress'}, {authentication: authenticationMetadata});
await timeout(125);
console.log(await taskBoardApplicationServices.findTaskBoardTasks(
  {taskBoardId}, {authentication: authenticationMetadata}));
```

At this point, the Sample Application implementation represents a fully functional software that is primarily missing a user interface. The next step is to improve the software architecture by separating the write concerns from the read concerns.

Chapter 11: Command Query Responsibility Segregation

Command Query Responsibility Segregation (CQRS) is an architectural pattern that separates a software into a write side and a read side. Typically, use cases that perform write operations differ in requirements and technical characteristics from the ones that primarily read information. These differences can include associated data and its structure, requirements for availability, consistency and performance, and even underlying code complexity. Still, many systems treat both use case types in a very similar way. They are processed by the same set of components and share a single data storage. Applying CQRS helps to clearly differentiate between write and read concerns and enables to evolve and scale them independently. Furthermore, it promotes to isolate the more critical write part of a Domain Model implementation.



Relation between CQRS and CQS

The architectural pattern CQRS is partially related to the design pattern CQS. To some extent, CQRS is CQS on a higher abstraction level. Initially, it was even referred to by the same name. The introduction of the term “Command Query Responsibility Segregation” aimed to clearly differentiate between the two concepts.

Architectural overview

CQRS divides a software into a write side and a read side. Typically, the pattern is applied to a selected area, but not to a complete system. For DDD-based software, this can map to a context implementation. The concept itself does not dictate specific technological decisions, but only implies that write and read concerns are somehow separated. Still, in many cases the two sides employ individual data storages and establish a synchronization mechanism. On top of that, they can apply custom concepts, run in different processes and even use distinct technologies. While a one-to-one alignment of both sides can be useful, it is not mandatory.

One write side may affect multiple read aspects and one query functionality can aggregate information from several write sides.



Separate data storages

This chapter focuses on explaining and illustrating CQRS with separate data storages for the write side and the read side. While it is possible to use the same storage, employing separate ones makes most sense when working towards Event Sourcing.

High-level picture

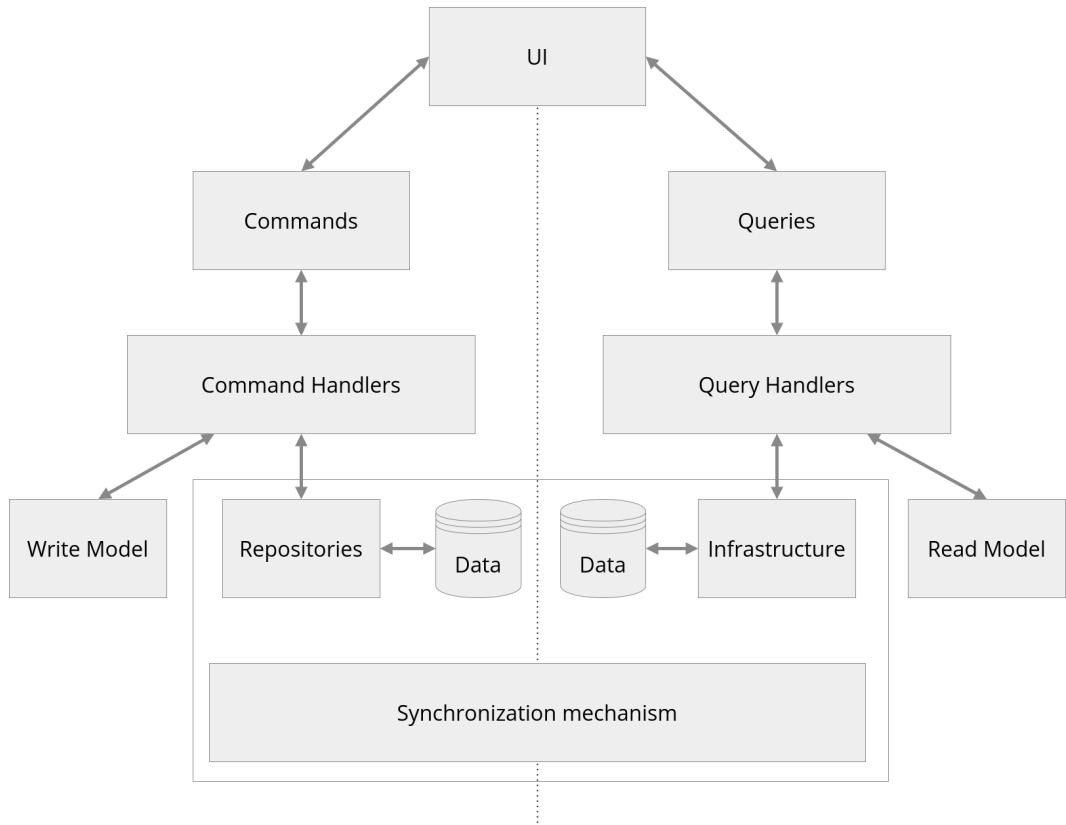


Figure 11.1: CQRS

The User Interface Layer is responsible for translating each meaningful user action into either a Command or a Query. The resulting messages are delivered to their responsible handler, which belongs to the Application Layer. Command Handlers load data via Repositories, construct Write Model components and execute a target action with the provided arguments. When encountering an error, the overall process is aborted. In case of success, the resulting state change is persisted. The synchronization mechanism consumes Write Model state and forwards it to domain-specific transformations that produce Read Model data. This information can be persisted with Repositories, but may also be saved with other storage mechanisms. Query Handlers load data based on given parameters, optionally construct domain-specific components and return a result.



Dependencies between write side and read side

Generally speaking, the two parts of a CQRS-based software should have minimal interdependencies. Naturally, a read side always depends on a write side for deriving its Read Models. In contrast, a write side should not depend on the other part. However, [Young] describes that “there are times where you will have to have the write side query the read side”.

Implicit CQRS

Many software projects implicitly apply CQRS without consciously aiming for it. This is because it often makes sense to execute different types of use cases in different ways. As explained earlier, it can be about data structures, consistency models, performance aspects or associated code complexity. Even the sole use of a caching mechanism can be interpreted as some form of CQRS. While Command Handlers operate on current state, the according Query Handlers respond with cached data. Another common scenario is when a software provides reporting functionalities. Typically, this is achieved by exposing preprocessed and aggregated data from a specialized storage. As example, consider a large online software platform that determines its number of users. Most certainly, this is not done by querying a production database.

Write and Read Model

The architectural segregation of write and read concerns typically causes the Domain Model implementation to be split into two parts. [Vernon, p. 139] explains that “we’d normally see

Aggregates with both command and query methods". When applying CQRS, the implementation for these two concerns is separated from each other. All the actions that cause a state to change are placed in the write part. In contrast, the data and structures that are used for querying and displaying information belong to the read side. The associated persistence-related mechanisms are optionally split and direct to individual data storages. Note that the conceptual Domain Model itself is not strictly required to reflect this technological separation. Nevertheless, in most cases it makes sense to account for this aspect.



Different terminologies

The Write Model and the Read Model can alternatively be called Command Model and Query Model. However, the terms Command and Query describe concepts that are part of the Application Layer. As the model implementation belongs to the Domain Layer, the use of distinct terms helps to differentiate between those areas.

Write Model

The **Write Model** and its implementation consist of all the actions that result in state changes. Effectively, a Write Model component is an Aggregate that contains only command operations. Consequently, the enclosed structure can be an arbitrary combination of Entities and Value Objects with one designated root. The associated Repository component must exclusively provide the means to load an instance via identifier and to save an Aggregate. Additional querying capabilities are not required. Overall, the write part of a Domain Model implementation deals with transactions, consistency and invariants. All the contained actions always operate on current state. For many systems, this is the more critical part of a Domain Model. Therefore, it can make sense to focus the majority of development efforts on this area.



Accommodations for persistence

Theoretically, a Write Model component can exclusively provide behavioral functionalities and fully encapsulate its own state. However, in many cases, it is still required to expose information as public attributes or getter functions for persistence purposes. Depending on the used persistence mechanism, this aspect can deteriorate the expressiveness of a Domain Model implementation.

Read Model

The **Read Model** and its implementation enclose all structures related to querying and displaying information. [Vernon, p. 141] describes this part as “de-normalized data model” that “is not meant to deliver domain behavior”. The information can be derived and aggregated from multiple write sides. While the underlying abstractions may manifest as concrete implementation components, they are not required to. The knowledge can also exclusively be expressed as structural description, such as a database schema. Read Models may be persistent, but can also be kept completely in memory. The latter approach demands the ability to rebuild them, such as when restarting a software. In contrast to the Write Model, the read part is not required to reflect the current state. Rather, it is expected to be eventually consistent.



Domain behavior on the read side?

Read Models are plain data structures and must not contain any domain-related behavior. Nevertheless, their creation and their querying may require the execution of specialized transformations, such as a calculation. This kind of behavior should be encapsulated in dedicated components that are part of the Domain Layer.

Example: Product reviews

Consider implementing the Domain Model and the persistence components for a product review feature. The functionality is meant to be integrated as part of an existing software. Its use cases consist of creating products, adding and updating reviews, and determining the average rating for a product. The Domain Model contains two Entity types. One is the product, which encloses an identity, a name and a category. The category is represented as string. All the attributes are defined immutable. The second part is the review, which consists of an identifier, a product identity, a rating and an optional comment. For the rating, a number between 0 and 5 can be assigned. Other than the identity and the product identifier, the rating and the comment are mutable.

The first two examples show the code for the review and the product Entity classes:

Product reviews: Product Entity

```
class Product {  
  
    id; name; category;  
  
    constructor({id, name, category}) {  
        Object.defineProperties(this, {  
            id: {value: id, writable: false},  
            name: {value: name, writable: false},  
            category: {value: category, writable: false},  
        });  
    }  
  
}
```

Product reviews: Review Entity

```
class Review {  
  
    id; productId; comment; #rating;  
  
    constructor({id, productId, rating, comment = ''}) {  
        Object.defineProperty(this, 'id', {value: id, writable: false});  
        Object.defineProperty(this, 'productId', {value: productId, writable: false});  
        this.rating = rating;  
        this.comment = comment;  
    }  
  
    set rating(rating) {  
        if (typeof rating != 'number' || rating < 0 || rating > 5)  
            throw new Error('invalid rating');  
        this.#rating = rating;  
    }  
  
    get rating() { return this.#rating; }  
  
}
```

The next code implements a dedicated service for calculating the average rating of a product ([run code usage](#)):

Product reviews: Rating calculator service

```
const ratingCalculator = {
    getAverageRating: ratings => {
        if (ratings.length === 0) return 0;
        return ratings.reduce((sum, rating) => sum + rating, 0) / ratings.length;
    },
};
```

The class `Product` represents the product Entity type. Its constructor accepts arguments for an identifier, a name and a category. All variables are assigned to according properties. The class `Review` expresses the Domain Model concept of a review. Again, all the passed in constructor arguments are assigned to their corresponding fields. However, the rating is defined as a private property together with according accessors. The setter function ensures that every passed in value is a valid number within the allowed range. The Domain Service `ratingCalculator` implements the operation `getAverageRating()` to calculate the average rating from a list of ratings. The combination of the three components enables to cover all previously defined use cases. Consequently, they represent a complete expression of the Domain Model.

The next examples implement the Repository components for both Entity types:

Product reviews: Product Repository

```
class ProductRepository extends ConcurrencySafeFilesystemRepository {

    constructor({storageDirectory}) {
        super({storageDirectory,
            convertToData: ({id, name, category}) => ({id, name, category}),
            convertToEntity: data => new Product(data)}),
    }
}
```

Product reviews: Review Repository

```
class ReviewRepository extends ConcurrencySafeFilesystemRepository {  
  
    constructor({storageDirectory}) {  
        super({storageDirectory,  
            convertToData:  
                ({id, productId, comment, rating}) => ({id, productId, comment, rating}),  
            convertToEntity: data => new Review(data))};  
    }  
  
    async findReviewsForProduct(productId) {  
        const files = await readdir(this.storageDirectory);  
        const ids = files.map(filename => filename.replace('.json', ''));  
        const reviews = await Promise.all(ids.map(id => this.load(id)));  
        return reviews.filter(review => review.productId === productId);  
    }  
}
```

The class `ProductRepository` is responsible for the persistence of product Entities. As the Entities are only accessed by their identifier, there is no need for an additional custom query. The class `ReviewRepository` provides the persistence mechanism for review Entities. In contrast to the other Repository, it implements the specialized query `findReviewsForProduct()`. The operation yields all review Entities associated with an individual product. For this purpose, it expects a product identifier as argument. As first step, all persisted review Entities are loaded. Then, the list is filtered by the given product identity. Finally, the matching items are returned. In combination with the rating calculator service, the average rating of a given product can be determined.

The next code shows an exemplary usage of all previously introduced components ([run code](#)):

Product reviews: Usage without CQRS

```
const productRepository = new ProductRepository(productRepositoryConfig);
const reviewRepository = new ReviewRepository(reviewRepositoryConfig);

const laptop = new Product({id: generateId(), name: 'T400', category: 'Laptop'});
await productRepository.save(laptop);

const negativeReviewId = generateId();
await reviewRepository.save(
  new Review({id: generateId(), productId: laptop.id, rating: 5}));
await reviewRepository.save(new Review({id: negativeReviewId,
  productId: laptop.id, rating: 5, comment: 'did not work at all!'}));

const reviews = await reviewRepository.findReviewsForProduct(laptop.id);
const averageRating = ratingCalculator.getAverageRating(
  reviews.map(({rating}) => rating));
console.log(`#${laptop.name} (rated ${averageRating})`);

const negativeReview = await reviewRepository.load(negativeReviewId);
negativeReview.rating = 4;
negativeReview.comment = 'got replacement, works fine!';
await reviewRepository.save(negativeReview);

const updatedReviews = await reviewRepository.findReviewsForProduct(laptop.id);
const updatedAverageRating = ratingCalculator.getAverageRating(
  updatedReviews.map(review => review.rating));
console.log(`#${laptop.name} (rated ${updatedAverageRating})`);
```

The code starts with instantiating Repository components for products and for reviews. This is followed by creating an exemplary product with the category “Laptop”. As next step, two new reviews are added. One of them has a low rating and an additional comment. Afterwards, all existing reviews for the product are retrieved via the query `findReviewsForProduct()`. The returned items are used to calculate the average rating using the service operation `getAverageRating()`. Then, the review with the low rating is updated and saved. Finally, the query `findReviewsForProduct()` is invoked again, and the average rating is recalculated and logged. While this implementation works, it has a poor performance, as it requires to always load all reviews for a product. This problem can be mitigated by applying CQRS.

The next example shows a Read Model storage component for product ratings ([run code usage](#)):

Product reviews: Product ratings

```
class ProductRatings {  
  
    #storage;  
  
    constructor({storageDirectory}) {  
        this.#storage = new JSONFileStorage(storageDirectory);  
    }  
  
    async addProduct(productId) {  
        await this.#storage.save(productId, {ratings: {}});  
    }  
  
    async addOrUpdateRating({productId, reviewId, rating}) {  
        const product = await this.#storage.load(productId);  
        product.ratings[reviewId] = rating;  
        product.averageRating = ratingCalculator.getAverageRating(  
            Object.values(product.ratings));  
        await this.#storage.save(productId, product);  
    }  
  
    async getAverageRating(productId) {  
        return (await this.#storage.load(productId)).averageRating;  
    }  
}
```

The class `ProductRatings` maintains persistent Read Models of product ratings and their average value. For the actual persistence, it uses the component `JSONFileStorage` as introduced in Chapter 8. Adding a product is done with the command `addProduct()`. The function `addOrUpdateRating()` is responsible for adding or updating a rating entry. First, it loads the ratings for the given product identity. Then, it either updates an existing rating entry or adds a new one. Next, the new average rating is calculated via the service. Finally, the updated data is stored again. The query function `getAverageRating()` is responsible for determining the average rating value for a specific product. This is done by loading the product entry for a given identifier and returning the pre-calculated attribute.



Where are the Read Model components?

As explained earlier, the read part of a Domain Model is not required to be expressed as concrete implementation components. This is because it consists of derived and potentially disposable data without domain-specific behavior. The necessary structures can be described in different ways, be it as persistence schema or through the shape of objects.

The final code shows an exemplary usage of the implementation with CQRS ([run code](#)):

Product reviews: Usage of CQRS variant

```
const productRepository = new ProductRepository(productRepositoryConfig);
const reviewRepository = new ReviewRepository(reviewRepositoryConfig);
const productRatings = new ProductRatings(productRatingsConfig);

const createProduct = async ({id, name, category}) => {
    await productRepository.save(new Product({id, name, category}));
    await productRatings.addProduct(id);
};

const createReview = async ({id, productId, rating, comment = ''}) => {
    await reviewRepository.save(new Review({id, productId, rating, comment}));
    await productRatings.addOrUpdateRating({reviewId: id, productId, rating});
};

const productId = generateId(), productName = 'T400';
const negativeReviewId = generateId();

await createProduct({id: productId, name: productName, category: 'Laptop'});

await createReview({id: generateId(), productId, rating: 5});
await createReview({id: negativeReviewId,
    productId, rating: 1, comment: 'did not work at all'});

const averageRating = await productRatings.getAverageRating(productId);
console.log(` ${productName} (rated ${averageRating})`);

const negativeReview = await reviewRepository.load(negativeReviewId);
Object.assign(negativeReview, {rating: 4, comment: 'got replacement, works fine'});
await reviewRepository.save(negativeReview);
await productRatings.addOrUpdateRating(
    {reviewId: negativeReview.id, productId, rating: negativeReview.rating});
```

```
const updatedAverageRating = await productRatings.getAverageRating(productId);
console.log(`#${productName} (rated ${updatedAverageRating})`);
```

The code starts with creating Repository instances for products and for reviews. Both components are reduced to their constructor without custom queries. Next, the class `ProductRatings` is instantiated. Then, the helper functions `createProduct()` and `createReview()` are defined. Each of them is responsible for creating a respective Entity, persisting it and updating the Read Model. The actual use case code is equivalent to the first approach. Determining the average product rating is done by executing the function `getAverageRating()` of the product ratings component. The performance issue from the previous implementation is eliminated. For determining the average rating, only a single file must be loaded. The manual update of a Read Model is one of multiple synchronization approaches. The most common ones are explained in the next section.

Read Model synchronization

The separation of the Write Model and the Read Model including their data storages necessitates a synchronization mechanism. Every change to a write side must be distributed to all derived read parts. The synchronization should happen in a non-transactional and asynchronous way. As explained previously, Read Models must be expected to be eventually consistent. While the update process may take time, it should commonly complete within seconds. Independent of the duration, a synchronization must always be reliable. The facilitation of this mechanism is a combination of infrastructural functionalities and Application Layer components. However, the specialized operations to transform Write Model knowledge into Read Model data architecturally belong to the Domain Layer. Depending on the complexity, it can make sense to express them as dedicated components.



Consistency delay in User Interfaces

Delays in consistency can introduce challenges for User Interfaces. Displayed information might become stale after issuing Write Model changes. This problem can be solved in different ways. [Vernon, p. 146] describes the possibility “to temporarily display the data that was submitted”. Alternatively, a User Interface can be notified of changes and update its contained information.

There are multiple possibilities for triggering Read Model updates in response to Write Model changes. The following subsections describe and illustrate the most common technical

synchronization approaches. This is preceded by the introduction of an example topic, which is used for the implementation of each individual approach.

Example: Song recommendation

Consider implementing the Domain Model for a song recommendation mechanism as part of a music streaming software. The use cases are to like songs, remove likes and to determine the most popular song per genre. Altogether, there are two individual components involved in this functionality. Both of them are greatly simplified for the sake of the example. The first part is the song, which consists of an identifier, an artist identity, a name and a genre. None of the attributes can be changed after creation. Also, the genre is represented as simple string. The second component is the user, which encloses an identifier, an e-mail and a collection of liked song identities. While the identifier and the e-mail are immutable, the referenced songs can change.



In-memory data instead of persistence

The following example implementations store all data directly in memory instead of using actual persistence. This reduces the overall code complexity and keeps the focus on the concepts to illustrate.

The first examples show the implementations of the Entity types for songs and users ([run code usage](#)):

Song recommendation: Song Entity

```
class Song {  
    id; artistId; name; genre;  
  
    constructor({id, artistId, name, genre}) {  
        Object.defineProperty(this, 'id', {value: id, writable: false});  
        Object.defineProperty(this, 'artistId', {value: artistId, writable: false});  
        Object.defineProperty(this, 'name', {value: name, writable: false});  
        Object.defineProperty(this, 'genre', {value: genre, writable: false});  
    }  
}
```

Song recommendation: User Entity

```
class User {

  id; email; #likedSongs = [];

  constructor({id, email}) {
    Object.defineProperty(this, 'id', {value: id, writable: false});
    Object.defineProperty(this, 'email', {value: email, writable: false});
  }

  addLike(songId) { this.#likedSongs.push(songId); }

  removeLike(songId) {
    this.#likedSongs.splice(this.#likedSongs.indexOf(songId), 1);
  }

  get likedSongs() { return this.#likedSongs.slice(); }

}
```

The next code provides a usage example that serves as blueprint for the interfaces of the subsequent synchronization approaches:

Song recommendation: Abstract usage example

```
const userId1 = generateId(), userId2 = generateId();
const songId1 = generateId(), songId2 = generateId();

createUser({id: userId1, email: 'john@example.com'});
createUser({id: userId2, email: 'jane@example.com'});
createSong(
  {id: songId1, artistId: generateId(), name: 'Yellow Submarine', genre: 'Rock'});
createSong(
  {id: songId2, artistId: generateId(), name: 'Stairway to Heaven', genre: 'Rock'});

addLike({userId: userId1, songId: songId1});
addLike({userId: userId1, songId: songId2});
addLike({userId: userId2, songId: songId1});
addLike({userId: userId2, songId: songId2});
removeLike({userId: userId1, songId: songId1});

setTimeout(() => { console.log(getMostLikedSong('Rock'))});
```

The code starts with generating identifiers for two users and two songs. Next, it creates and saves the according user objects via the function `createUser()`. This is followed by instantiating and saving two song Entities with the operation `createSong()`. Afterwards, the command `addLike()` is invoked four times, resulting in adding one like per user to each song. Then, the operation `removeLike()` is invoked to remove a single like again. Finally, a timeout is requested, in which the most liked song of the according category is logged. When executing this code with one of the following synchronization approaches, the query `getMostLikedSong()` should return the second song. While both Entities are equally popular at some point, the number of likes for the first song is decreased again.

Pushing updates

One synchronization approach is to actively trigger Read Model updates from the write side. Whenever a state change is persisted successfully, all derived data structures are requested to be updated. Typically, this is done by forwarding all modified values. As explained earlier, the process should only happen asynchronously. In general, this approach makes sense for smaller systems and when first introducing CQRS. Its implementation is straight forward and results in a low consistency delay. However, for most software products, it is not a long-term solution. The key disadvantage is that the write side becomes dependent on each downstream read part. This is because it must know which data structures to update and how this is achieved. Also, the synchronization requires custom efforts to ensure reliability.

The following code example provides a component that maintains Read Models of songs and their likes ([run code usage](#)):

Song recommendation: Most liked songs component

```
class MostLikedSongs {  
  
    #songs = new Map(); #mostLikedSongByGenre = new Map();  
  
    addSong({id, name, genre}) {  
        this.#songs.set(id, {id, name, genre, likes: 0});  
    }  
  
    addLike(songId) {  
        const song = this.#songs.get(songId);  
        song.likes += 1;  
        const mostLikedSong = this.#mostLikedSongByGenre.get(song.genre);  
        if (!mostLikedSong || song.likes >= mostLikedSong.likes)  
            this.#mostLikedSongByGenre.set(song.genre, song);  
    }  
}
```

```

}

removeLike(songId) {
  const song = this.#songs.get(songId);
  song.likes -= 1;
  const mostLikedSong = this.#mostLikedSongByGenre.get(song.genre);
  if (song.id === mostLikedSong.id) {
    const songList = Array.from(this.#songs.values());
    const songsWithSameGenre = songList.filter(({genre}) => genre === song.genre);
    const newMostLiked = songsWithSameGenre.sort((a, b) => b.likes - a.likes)[0];
    this.#mostLikedSongByGenre.set(song.genre, newMostLiked);
  }
}

getMostLikedSong(genre) { return this.#mostLikedSongByGenre.get(genre); }

}

```

The class `MostLikedSongs` enables to determine the most liked song per genre. For this purpose, it maintains two Read Models, both implemented as `Map` instances. While the object `songs` contains all songs together with their likes, the most popular ones are stored redundantly in `mostLikedSongByGenre`. Adding a new entry is done with the operation `addSong()`. The command `addLike()` retrieves the song for a given identity and adds a like. Also, it updates the most popular one if the changed song has at least the same like count. The command `removeLike()` retrieves a song entry and decrements its likes. Whenever this affects the most liked item in a genre, the operation re-evaluates this aspect. Finally, the query `getMostLikedSong()` returns the current favorite for a given genre.



Mixing of Infrastructure and Domain

The component for maintaining Read Models in the example partially mixes infrastructural concerns with domain-related aspects. Generally speaking, the two parts should be separated from each other. At the same time, it is advisable to be pragmatic when dealing with rather trivial or generic Read Model transformations.

The next implementation provides the operations for all use cases with interfaces that match the previously shown abstract usage code ([run code usage](#)):

Song recommendation: Push updates

```
const songDatabase = new Map();
const userDatabase = new Map();
const mostLikedSongs = new MostLikedSongs();

const createUser = ({id, email}) => {
  userDatabase.set(id, new User({id, email}));
};

const createSong = ({id, artistId, name, genre}) => {
  songDatabase.set(id, new Song({id, artistId, name, genre}));
  setTimeout(() => mostLikedSongs.addSong({id, name, genre}));
};

const addLike = ({userId, songId}) => {
  userDatabase.get(userId).addLike(songId);
  setTimeout(() => mostLikedSongs.addLike(songId));
};

const removeLike = ({userId, songId}) => {
  userDatabase.get(userId).removeLike(songId);
  setTimeout(() => mostLikedSongs.removeLike(songId));
};

const getMostLikedSong = genre => mostLikedSongs.getMostLikedSong(genre);
```

The code starts with creating `Map` objects as databases for songs and users. This is followed by instantiating the `MostLikedSongs` class. Next, all use case operations are implemented. The function `createUser()` instantiates and persists a user Entity. Creating and saving a song is done with the command `createSong()`. This function also executes the operation `addSong()` of the `MostLikedSongs` instance. The commands `addLike()` and `removeLike()` add or remove a liked song for a user. Internally, they both call the matching update action on the `MostLikedSongs` instance. Finally, the query `getMostLikedSong()` returns the most liked item in a genre. Overall, this approach provides a good performance. One disadvantage is that the write side depends on the read side. Also, the implementation fails to guarantee a reliable synchronization.

On-demand computation

Another possibility for synchronization is to lazily compute Read Models. Upon requesting, the required write parts are accessed and all derived data is computed. The amount of

total computations can be reduced by caching the results. This approach has multiple advantages. For one, any Read-Model-related processing is deferred until needed. Secondly, when combined with caching, the consistency delay can be configured. One disadvantage is that every first request for a data set has an increased latency. For complex computations, this might not even be feasible. Compared to the first approach, the write side stays independent of the read part. However, it must provide querying capabilities to expose required information. Also, the reliability is slightly better, as Read Model computations can be retried when failing.

The first example shows a helper operation to create a cached version of a function ([run code usage](#)):

Song recommendation: Create cached function

```
const createCachedFunction = (originalFunction, cacheDurationMs) => {
  const cache = {};
  return function cachedExecution() {
    const currentTime = Date.now(), key = JSON.stringify(arguments);
    if (cache[key] && currentTime - cache[key].time <= cacheDurationMs)
      return cache[key].result;
    cache[key] = {time: currentTime, result: originalFunction(...arguments)};
    return cache[key].result;
  };
};
```

The operation `createCachedFunction()` creates a cached version of a function that skips invocations for equal inputs within a given duration. As arguments, it expects an original function and a cache duration in milliseconds. The return value is a decorated function, that closes over a mutable cache object. Whenever the returned operation is invoked, it first checks whether the cache contains an entry with the given arguments. If an entry exists and its age does not exceed the cache duration, the previously stored result is returned immediately. Otherwise, the original operation is executed and a new cache entry is created before returning the result to the caller. This utility function provides the ability to add caching behavior to any operation.

The next code shows the implementation of the defined use cases with an on-demand Read Model computation ([run code usage](#)):

Song recommendation: On-demand computation

```
const songDatabase = new Map();
const userDatabase = new Map();

const createUser = ({id, email}) => {
    userDatabase.set(id, new User({id, email}));
};

const createSong = ({id, artistId, name, genre}) => {
    songDatabase.set(id, new Song({id, artistId, name, genre}));
};

const addLike = ({userId, songId}) => {
    userDatabase.get(userId).addLike(songId);
};

const removeLike = ({userId, songId}) => {
    userDatabase.get(userId).removeLike(songId);
};

const getMostLikedSong = createCachedFunction(genre => {
    console.log('computing most liked song');
    const songs = Array.from(songDatabase.values());
    const relevantSongs = songs.filter(song => song.genre === genre);
    return relevantSongs.sort((b, a) => b.likes - a.likes)[0];
}, 1000);
```

The code also starts with creating two Map objects as databases for both Entity types. All the use case operations related to state changes exclusively update the Write Model. In contrast, the query `getMostLikedSong()` computes the most liked song for a given genre on demand. The operation retrieves all songs from the database, filters by affected genre, sorts by likes and returns the first item. Caching is enabled by decorating the operation with the helper `createCachedFunction()`. Overall, this approach has a low complexity. The main disadvantage is the poor read performance. Determining the most liked song for a genre requires looping through all song Entities. While the caching helps to reduce subsequent computations, the approach is not feasible for a larger number of persistent songs.

Event notifications

The third synchronization approach is to use Event notifications for triggering Read Model updates. This especially makes sense when the implementation is based on DDD and incor-

porates Domain Events or applies Event Sourcing. For each state change that is successfully persisted, all related events are distributed. The synchronization mechanism for associated Read Models receives the event notifications and requests to update the derived data structures. This approach has multiple advantages. For one, the write part stays completely agnostic of any read concerns. Also, every read part receives all required information without the need to actively query for it. Furthermore, since the event distribution should provide a guaranteed delivery, the synchronization can also be considered reliable. One potential downside of this approach is the additional complexity.

The following code implements the specified use case operations with Domain Event notifications to trigger the Read Model synchronization ([run code usage](#)):

Song recommendation: Event notifications

```
const SongCreatedEvent = createEventType(
  'SongCreated', {songId: 'string', name: 'string', genre: 'string'});
const UserLikedSongEvent = createEventType(
  'UserLikedSong', {userId: 'string', songId: 'string'});
const UserRemovedSongLikeEvent = createEventType(
  'UserRemovedSongLike', {userId: 'string', songId: 'string'});

const songDatabase = new Map();
const userDatabase = new Map();
const mostLikedSongs = new MostLikedSongs();

const createUser = ({id, email}) => userDatabase.set(id, new User({id, email}));

const createSong = ({id, artistId, name, genre}) => {
  songDatabase.set(id, new Song({id, artistId, name, genre}));
  eventBus.publish(new SongCreatedEvent({songId: id, name, genre}));
};

const addLike = ({userId, songId}) => {
  userDatabase.get(userId).addLike(songId);
  eventBus.publish(new UserLikedSongEvent({userId, songId}));
};

const removeLike = ({userId, songId}) => {
  userDatabase.get(userId).removeLike(songId);
  eventBus.publish(new UserRemovedSongLikeEvent({userId, songId}));
};

const eventHandlers = {
  SongCreated: ({data: {songId, name, genre}}) =>
```

```
mostLikedSongs.addSong({id: songId, name, genre}),  
UserLikedSong: ({data}) => mostLikedsongs.addLike(data.songId),  
UserRemovedSongLike: ({data}) => mostLikedsongs.removeLike(data.songId),  
};  
  
[SongCreatedEvent, UserLikedSongEvent, UserRemovedSongLikeEvent].forEach(Event =>  
  eventBus.subscribe(Event.type, eventHandlers[Event.type]));  
  
const getMostLikedSong = genre => mostLikedsongs.getMostLikedSong(genre);
```



Event creation in the Application Layer

For the example, the events are created and published directly from within the Application Layer. The goal is to leave the previously introduced Entity classes unmodified. Normally, the creation of events should only happen from within Domain Model components.

The code starts with creating the event types `SongCreatedEvent`, `UserLikedSongEvent` and `UserRemovedSongLikeEvent`. Next, both Entity databases are created. Afterwards, the class `MostLikedsongs` is instantiated, which is the same component as for the first approach. Then, the write-related use case operations are implemented. Each of them creates and publishes an according event. Updating the Read Model is done through the object `eventHandlers`. For each event type, it defines a function that performs the necessary update. These operations are registered as subscribers on the Event Bus. Finally, the query `getMostLikedSong()` returns the most popular song for a given genre. Overall, this approach is the most powerful variant. Event distribution enables a loosely coupled and reliable way of synchronization. One implication is the need to introduce event types.



Domain Events are Write Model concerns

When applying CQRS and strictly separating a Domain Model implementation, Domain Events can only occur from within the Write side. Since they represent something meaningful that happened in a Domain, they must always relate to an actual state change.

Rebuilds

Depending on the synchronization mechanism and the use of persistence, Read Models may need to be rebuilt at certain points. Especially when derived data structures are exclusively

kept in memory, there must be a possibility to recreate them completely. One common scenario is when a software needs to be restarted. Even more, there are scenarios where persistent Read Models need to be recomputed. For example, when the transformation mechanism for a de-normalized data structure contains errors. Also, when additional information is added retroactively to Write Model components, rebuilding related Read Models can be necessary. In general, there is no silver bullet for implementing an adequate mechanism to enable this functionality. However, it commonly requires to expose querying capabilities on the related write parts.

Commands and Queries

Commands and **Queries** represent the intent to execute an individual use case with a certain set of arguments. Commands are meant to trigger actions that cause a state to change, for example when a user wants to login. They affect the write side of a Domain Model implementation. In contrast, Queries intend to retrieve information, such as determining the number of active users. Consequently, they target Read Model data. Architecturally speaking, both belong in the Application Layer and are not part of the Domain. While they represent different use case types, their anatomy and processing is similar. As with Domain Events, the purpose is to encapsulate meaningful knowledge. The difference is that events describe past occurrences, while Commands and Queries represent something that should happen.

Naming conventions

The type names for Commands and Queries must be chosen with care. In general, each one should describe the respective use case to execute. Furthermore, the selected terms should be expressive and ideally apply the prevailing Ubiquitous Language. One useful pattern is to combine the action to execute in present tense with the primarily affected subject. For Commands, the subject is the associated Write Model component and the action is the operation to invoke. For Queries, the subject is typically the type of data to be retrieved. The action is either a single operation or a process, which can be substituted with verbs like “get” or “find”. Whenever a use case does not align one-to-one with Domain Model components, the naming should express this circumstance.

The following tables show Command and Query naming examples, of which some adhere to the general recommendation:

Exemplary Command names

Component	Action	Potential Command type name
browser tab	reload()	ReloadBrowserTab
meeting	addParticipant()	AddMeetingParticipant
guestbook	writeMessage()	WriteGuestbookMessage
user	login()	LoginUser
newsletter	subscribe()	SubscribeToNewsletter

Exemplary Query names

Component	Action	Potential Query type name
browser	determine active tab	DetermineActiveBrowserTab
meeting	get all participants	ListMeetingParticipants
guestbook	find message by criteria	FindGuestbookMessage
user	get login status	GetUserLoginStatus
newsletter	count subscriptions	CountNewsletterSubscriptions

Structure and content

The structure and the content of Commands and Queries is similar to Domain Events. In general, they should consist of immutable attributes that can be serialized and are easy to consume. Conceptually, they also contain both data and metadata. The data part must at least include a type name. On top of that, any use-case-related information can be enclosed. Most commonly, this includes the identity of the primarily affected subject. For the metadata part, the most important information is a unique message identifier. Besides the optional time of occurrence, Commands and Queries also commonly incorporate authentication information for identity verification and authorization checks. Despite the structural similarities with Domain Events, the two concepts should be clearly separated, as they represent different architectural aspects.

Typical Command and Query contents

Information	Category	Description
type	data	Name of message Type
subject id	data	Identity of affected subject
custom data	data	Additional use case parameters
id	metadata	Unique event identifier
creation time	metadata	Timestamp of event occurrence

Typical Command and Query contents

Information	Category	Description
authentication	metadata	Authentication information

The following code provides a generic factory to create specialized Command and Query types:

Messages: Message Type factory

```

let generateId = () => '';
let createDefaultMetadata = () => ({});

const createMessageType = (type, dataStructure) =>
  function Message({data, metadata = {}}) {
    const containsInvalidFields = Object.keys(dataStructure).some(property => {
      const dataTypes = [].concat(dataStructure[property]);
      return dataTypes.every(dataType => typeof data[property] != dataType);
    });
    if (containsInvalidFields) throw new TypeError(
      `expected data structure: ${JSON.stringify(dataStructure, null, 2)}`
    );
    Object.assign(this, {type, data, id: generateId(),
      metadata: {...createDefaultMetadata(), ...metadata}});
  };
}

const setIdGenerator = idGenerator => {
  generateId = idGenerator;
};

const setDefaultMetadataProvider = defaultMetadataProvider => {
  createDefaultMetadata = defaultMetadataProvider;
};

const messageTypeFactory =
  {createMessageType, setIdGenerator, setDefaultMetadataProvider};

```

The usage of this component can be seen in the next example ([run code](#)):

Messages: Message Type factory usage

```
const CreateUserCommand = createMessageType('CreateUser',
  {userId: 'string', email: 'string', username: ['string', 'undefined']}));
const GetUserQuery = createMessageType('GetUser', {userId: 'string'});

messageTypeFactory.setIdGenerator(generateId);
messageTypeFactory.setDefaultValueProvider(() => ({creationTime: new Date()}));

console.log(new CreateUserCommand(
  {data: {userId: '1', email: 'alex@example.com'}, metadata: {}}));
console.log(new CreateUserCommand(
  {data: {userId: '1', email: 'james@example.com', username: 'james'}}));
console.log(new GetUserQuery({data: {userId: '1'},
  metadata: {authentication: {subjectId: '123'}}}));
```

The component `messageTypeFactory` is a factory to create specialized Command and Query types with type-checked data. Its main operation `createMessageType()` expects a type name and a data structure. The return value is a constructor function for creating message instances. Overall, the functionality is almost identical to the event type factory. However, there is one key difference. In addition to the data, the returned constructor also accepts metadata. The argument is merged with the values returned from the function `createDefaultMetadata()`, which can be customized via `setDefaultMetadataProvider()`. This enables to provide automatically created generic metadata, but also pass in specialized values, such as authentication information. Although the component is similar to the event type factory, both functionalities should be strictly separated, as the concepts themselves.

Commands vs. Domain Events

The concepts Commands and Domain Events are commonly confused with each other. Despite their conceptual and structural overlaps, they serve different purposes. Effectively, a Command is an instruction to do something that is phrased in the present tense. The message is directed to a single receiver, which can either accept or reject the instruction. In contrast, a Domain Event is a description of an occurrence, phrased in the past tense. The message is typically broadcasted and can only be acknowledged by its receivers. One aspect that helps to decide which concept fits better for a specific case is an architectural classification. Domain Events must originate from the Domain Layer, while Commands are sent from the outside into the Application Layer.

Differences between Commands and Events

	Commands	Events
Purpose	Instruction to act	Description of occurrence
Naming	Present tense	Past tense
Communication	One-to-one	One-to-many
Possible actions	Accept/Reject	Acknowledge

Passive-aggressive Command



When a Command is mistakenly modeled as Domain Event, its type name typically reflects this circumstance. [Fowler] calls such a message a “passive-aggressive command”, as it seemingly describes an occurrence, but in fact is an instruction. This also means that the type name itself can serve as a hint to an underlying design issue.

Example: Post likes

Consider implementing a like mechanism as part of a social network. The required functionality is to add individual likes to a post. One solution approach is to introduce the Command type “LikePost”, which can either be accepted and processed or get rejected. Another option is to define the Domain Event type “PostLiked”. When a client sends such an event, it can only be acknowledged, but cannot be rejected. Since Domain Events should always only originate from the Domain part, the latter approach seems ill-suited. However, the judgement heavily depends on the architectural viewpoint. Assuming that the client space is a meaningful Domain in itself, publishing the event type “PostLiked” can be correct. As described previously, the best approach always depends on the respective scenario.

There are no one-way Commands



The idea of a one-way Command is an instruction to do something without the possibility to reject it. However, this concept fits more with the description of an event rather than a Command. The inability to reject an instruction inherently makes it a fact that can only be acknowledged.

Command and Query Handlers

Command and Query Handlers are responsible for processing Commands and Queries. Typically, they consume infrastructural functionalities and access Domain Model components to execute a use case. Architecturally speaking, they belong to the Application Layer. Even more, they are a specialized form of Application Services. Therefore, some recommendations and considerations apply equally. Each handler should be concerned with one use case and affect only a single transaction. The implementation can follow two different styles. While the “dedicated style” promotes a separate unit per handler, the “categorized style” groups multiple related operations into one component. Regardless of the respective style, Command Handlers and Query Handlers should normally not be mixed with each other. As with the Application Services in Chapter 10, this book favors the categorized style.

The following code provides a factory to create a type-based message forwarding mechanism:

Messages: Message forwarder factory usage

```
const createMessageForwarder = (target, {messageSuffix} = {messageSuffix: ''}) =>
  message => {
    const messageHandlerName = `handle${message.type}${messageSuffix}`;
    if (!target[messageHandlerName])
      throw new Error(`invalid message: ${JSON.stringify(message)}`);
    return target[messageHandlerName](message);
};
```

The usage of this functionality can be seen in the next example ([run code](#)):

Messages: Message forwarder factory usage

```
class CommandHandlers {

  handleCommand = createMessageForwarder(this, {messageSuffix: 'Command'});

  handleFirstUseCaseCommand(command) {
    console.log('executing first use case with', command.data);
  }

  handleSecondUseCaseCommand(command) {
    console.log('executing second use case with', command.data);
  }
}
```

```
}
```

```
const commandHandlers = new CommandHandlers();
commandHandlers.handleCommand({type: 'FirstUseCase', data: {foo: 'bar'}});
commandHandlers.handleCommand({type: 'SecondUseCase', data: {bar: 'foo'}});
```

The factory `createMessageForwarder()` creates an operation that forwards messages to functions on a target object based on their type. As arguments, it expects a target and an optional message suffix property. The return value is an operation that forwards a message upon invocation. For the target function name, it combines the prefix “handle” with the message type and the optional suffix. If the result points to an existing function, it is executed with the original message as argument. Otherwise, an exception is thrown. The example usage defines a class with categorized Command Handlers. Also, it creates a message forwarder mechanism and assigns it to the field `handleCommand`. This operation serves as single entry point for all messages to handle, independent of their type.

Example: Time Tracking

Consider implementing the Domain Model, the Application Layer and in-memory storages for a time tracking functionality. The purpose is to track activities and to create a time sheet of completed items, sorted by their start. The according Domain Model consists of three parts. One is the activity, which contains an identifier, a label, a start time and an end time. While all its attributes are immutable, the end time is only set at a later point. The second model part is the time tracking itself, which consists of an identity and a collection of activities. New activities can be started and existing ones can be stopped. Both actions should trigger event notifications to synchronize the third Domain Model part, which is the time sheet report.

The first example defines Domain Event types for starting and stopping an individual activity:

Time Tracking: Domain Events

```
const TimeTrackingActivityStartedEvent = createEventType(
  'TimeTrackingActivityStarted', {timeTrackingId: 'string',
    activityId: 'string', activityLabel: 'string', startTime: 'number'});

const TimeTrackingActivityStoppedEvent = createEventType(
  'TimeTrackingActivityStopped',
  {timeTrackingId: 'string', activityId: 'string', endTime: 'number'})
```

The next code implements the time tracking Entity with activities as contained anonymous objects ([run code](#)):

Time Tracking: Time Tracking Entity

```
class TimeTracking {

  id; #activitiesById = {}; #eventBus;

  constructor({id, eventBus}) {
    Object.defineProperty(this, 'id', {value: id, writable: false});
    this.#eventBus = eventBus;
  }

  startActivity({activityId, activityLabel, startTime}) {
    this.#activitiesById[activityId] = {label: activityLabel, startTime};
    this.#eventBus.publish(new TimeTrackingActivityStartedEvent(
      {timeTrackingId: this.id, activityId, activityLabel, startTime}));
  }

  stopActivity({activityId, endTime}) {
    this.#activitiesById[activityId].endTime = endTime;
    this.#eventBus.publish(new TimeTrackingActivityStoppedEvent(
      {timeTrackingId: this.id, activityId, endTime}));
  }
}
```

The following example provides the time sheet report Read Model ([run code](#)):

Time Tracking: Time Sheet

```
class TimeSheet {

    #activitiesById = {}; #completedActivities = [];

    recordActivityStart({activityId, activityLabel, startTime}) {
        this.#activitiesById[activityId] = {label: activityLabel, startTime};
    }

    recordActivityStop({activityId, endTime}) {
        const activity = this.#activitiesById[activityId];
        Object.assign(activity, {endTime, duration: endTime - activity.startTime});
        const indexToInsert = this.#completedActivities.findIndex(
            activity => activity.duration < activity.duration);
        this.#completedActivities.splice(indexToInsert, 0, activity);
    }

    getCompletedActivities() { return this.#completedActivities.slice(); }

}
```

The last code implements a class for event handlers that synchronizes the Read Model data upon Domain Event notifications:

Time Tracking: Read Model synchronization

```
class ReadModelSynchronization {

    #timeSheetStorage;

    constructor({timeSheetStorage, eventBus}) {
        this.#timeSheetStorage = timeSheetStorage;
        this.activate = () =>
            ['TimeTrackingActivityStarted', 'TimeTrackingActivityStopped'].forEach(
                type => eventBus.subscribe(type, this.handleEvent.bind(this)));
    }

    handleEvent = createMessageForwarder(this, {messageSuffix: 'Event'});

    async handleTimeTrackingActivityStartedEvent(event) {
        const {timeTrackingId, activityId, activityLabel, startTime} = event.data;
        if (!this.#timeSheetStorage.has(timeTrackingId))
            this.#timeSheetStorage.set(timeTrackingId, new TimeTrackingReport());
    }
}
```

```
    const timeSheet = this.#timeSheetStorage.get(timeTrackingId);
    timeSheet.recordActivityStart({activityId, activityLabel, startTime});
}

async handleTimeTrackingActivityStoppedEvent(event) {
    const {timeTrackingId, activityId, endTime} = event.data;
    const timeSheet = this.#timeSheetStorage.get(timeTrackingId);
    timeSheet.recordActivityStop({activityId, endTime});
}

}
```

The class `TimeSheet` represents the Read Model. Its commands `recordActivityStart()` and `recordActivityStop()` are the counterparts to the Write Model operations of the `TimeTracking` Entity type. Besides performing a state mutation, recording an activity stop also causes to update the list of completed items. For that, the operation determines the right index to insert the new activity. This is required as the Domain Model does not enforce any order of start and stop actions. The query operation `getCompletedActivities()` returns the collection of completed items. The class `ReadModelSynchronization` implements subscriber operations to create and synchronize time sheet Read Models. Its constructor expects an in-memory storage and an Event Bus instance. Also, the component uses the previously introduced helper function `createMessageForwarder()` to simplify the subscription code.

The illustrated code provides the implementation for the write side and the read side together with a synchronization mechanism. This example functionality is extended and completed throughout the following subsections that describe and illustrate Command and Query handling. For simplicity reasons, the following code uses plain objects for Commands and Queries instead defining specialized message types.

Command handling

Command Handlers process Commands and affect the write side of a software. Typically, they load selected Write Model components via Repositories, operate on them with given inputs and persist all state changes. When encountering an error, they should throw an exception. This can be due to an infrastructural aspect or a domain-specific concern, such as an invariant protection. Other than that, Command Handlers should have no return value. Nevertheless, simple operational status codes can be acceptable. As with any Application Service implementation, one individual service should at most affect a single transaction. There may be situations where processing a Command requires information from a Query

Handler. This type of dependency should ideally be avoided, as it makes a write side dependent on a read side.

The following code shows a class for the categorized Command Handlers of the previously introduced time tracking functionality:

Time Tracking: Command Handlers

```
class CommandHandlers {  
  
    #timeTrackingStorage; #eventBus;  
  
    constructor({timeTrackingStorage, eventBus}) {  
        this.#timeTrackingStorage = timeTrackingStorage;  
        this.#eventBus = eventBus;  
    }  
  
    handleCommand = createMessageForwarder(this, {messageSuffix: 'Command'});  
  
    async handleCreateTimeTrackingCommand(command) {  
        const {timeTrackingId} = command.data;  
        if (this.#timeTrackingStorage.has(timeTrackingId)) return;  
        this.#timeTrackingStorage.set(timeTrackingId,  
            new TimeTracking({id: timeTrackingId, eventBus: this.#eventBus}));  
    }  
  
    async handleStartTimeTrackingActivityCommand(command) {  
        const {timeTrackingId, activityId, activityLabel, startTime} = command.data;  
        const timeTracking = this.#timeTrackingStorage.get(timeTrackingId);  
        timeTracking.startActivity({activityId, activityLabel, startTime});  
    }  
  
    async handleStopTimeTrackingActivityCommand(command) {  
        const {timeTrackingId, activityId, endTime} = command.data;  
        const timeTracking = this.#timeTrackingStorage.get(timeTrackingId);  
        timeTracking.stopActivity({activityId, endTime});  
    }  
}
```

The usage of this part can be seen in the next example ([run code](#)):

Time Tracking: Command Handlers usage

```
const timeTrackingStorage = new Map();
const commandHandlers = new CommandHandlers({timeTrackingStorage, eventBus});

const timeTrackingId = generateId();
const activityId = generateId(), activityLabel = 'development';

const commands = [
  {type: 'CreateTimeTracking', data: {timeTrackingId}},
  {type: 'StartTimeTrackingActivity',
    data: {timeTrackingId, activityId, activityLabel, startTime: Date.now()}},
  {type: 'StopTimeTrackingActivity',
    data: {timeTrackingId, activityId, endTime: Date.now() + 1000 * 3600}},
];
for (const command of commands)
  await commandHandlers.handleCommand(command);
```

The class `CommandHandlers` implements all write-related use cases for the time tracking functionality. Its constructor expects a storage component and an Event Bus. Also, it creates a message forwarder mechanism and assigns it to the field `handleCommand`. The operation `handleCreateTimeTrackingCommand()` is responsible for creating a time tracking and saving it. Starting and respectively stopping an activity is the responsibility of the functions `handleStartTimeTrackingActivity()` and `handleStopTimeTrackingActivity()`. The usage example starts with instantiating a `Map` for saving time tracking instances. Next, it creates an instance of the component `CommandHandlers` with the storage component and an Event Bus as arguments. Afterwards, the code defines two identifiers and an activity label. Finally, an array of exemplary commands is created and issued to the Command Handlers in an asynchronous loop.

Misuse of commands

There are resources and technologies that promote the idea of passing Commands directly as arguments to Domain Layer components. In my opinion, this is problematic, especially with regard to Software Architecture. For one, Commands are a concept of the Application Layer and not the Domain part. Domain Model components should not be aware of their existence. Secondly, they are a description of a request to execute a use case. Consequently, Commands seem ill-suited as arguments for functions of domain-related components. Also, a Domain Model implementation can use rich data structures for function arguments, such as Value Objects. In contrast, a Command must contain simple data independent of such

structures. Finally, a Command may not even map to a specific Domain Layer component at all.

Query handling

Query Handlers respond to Queries and access the read part of a software. Their mechanics can vary depending on the applied patterns and used technologies. The main responsibility is to retrieve data based on given input criteria. Typically, this is done by either accessing a persistent storage or an in-memory representation. The stored data can be structured or unstructured, in which case it may be transformed into specialized Read Model components. The final result is returned to the Query sender. Errors that occur during the process should not be forwarded as exceptions. Ideally, they are translated into meaningful return values. Generally, Query Handlers should be free of side effects and must therefore not perform any state changes. Also, they should not depend on Command Handlers.

The following code shows a class for categorized Query Handlers of the time tracking implementation:

Time Tracking: Query Handlers

```
class QueryHandlers {  
  
    #timeSheetStorage;  
  
    constructor({timeSheetStorage}) {  
        this.#timeSheetStorage = timeSheetStorage;  
    }  
  
    handleQuery = createMessageForwarder(this, {messageSuffix: 'Query'});  
  
    async handleGetTimeSheetQuery(query) {  
        const {timeTrackingId} = query.data;  
        const timeSheet = this.#timeSheetStorage.get(timeTrackingId);  
        return timeSheet.getCompletedActivities();  
    }  
}
```

This is followed by a usage example ([run code](#)):

Time Tracking: Query Handlers usage

```
const timeTrackingStorage = new Map();

const commandHandlers = new CommandHandlers({timeTrackingStorage, eventBus});

const timeSheetStorage = new Map();

const readModelSynchronization =
  new ReadModelSynchronization({timeSheetStorage, eventBus});
const queryHandlers = new QueryHandlers({timeSheetStorage});
readModelSynchronization.activate();

const [timeTrackingId, activity1Id, activity2Id] =
  [generateId(), generateId(), generateId()];

const time1 = Date.now(), time2 = time1 + 1000, time3 = time2 + 2000;

const commands = [
  {type: 'CreateTimeTracking', data: {timeTrackingId}},
  {type: 'StartTimeTrackingActivity', data: {timeTrackingId,
    activityId: activity1Id, activityLabel: 'coding', startTime: time1}},
  {type: 'StopTimeTrackingActivity', data: {timeTrackingId,
    activityId: activity1Id, endTime: time2}},
  {type: 'StartTimeTrackingActivity', data: {timeTrackingId,
    activityId: activity2Id, activityLabel: 'testing', startTime: time2}},
  {type: 'StopTimeTrackingActivity', data: {timeTrackingId,
    activityId: activity2Id, endTime: time3}},
];
for (const command of commands)
  await commandHandlers.handleCommand(command);
await timeout(125);
const query = {type: 'GetTimeSheet', data: {timeTrackingId}};
console.log(await queryHandlers.handleQuery(query));
```

The component `QueryHandlers` provides the read-specific service functionality for the time tracking example. Its constructor expects a time sheet storage component. Even though the class only contains a single handler, it introduces a message forwarder operation for a universal interface. The operation `handleGetTimeSheetQuery()` retrieves a time sheet instance, invokes its operation `getCompletedActivities()` and returns the result. The usage code starts with defining a `Map` instance for time trackings and another one for time sheets. This is followed by instantiating the Command Handlers, Domain Event Handlers and Query

Handlers. Next, the code introduces example identifiers and timestamps. Then, an array of Commands is processed by the Command Handlers. Finally, a Query with the type “GetTimeSheet” is executed through the Query Handlers and its result is logged.

Sample Application: CQRS

This section illustrates the introduction of CQRS to the Sample Application implementation. Each of the individual context implementations requires adjustments to multiple parts of different architectural layers. Application Services are separated into Commands with Command Handlers and Queries with Query Handlers. Domain Models and their implementations are split into write and read parts and additional Domain Events are introduced. For the synchronization of Read Model data, dedicated components are implemented. Repositories and factories are adjusted accordingly. As overall approach, all Read Models are exclusively stored as in-memory representations. For brevity reasons, this section only shows the most distinct parts of the reworked source code. Specifically, the Command Handler classes are excluded, as they are very similar to the Application Services from the last chapter.

The following code example demonstrates the similarity between an Application Service class and a Command Handlers component:

User context: User constructor

```
class ApplicationService {  
    useCase(data, metadata) { /* .. */ }  
}  
  
class CommandHandlers {  
    handleUseCaseCommand(command) { /* .. */ }  
}
```



What about Domain Event handlers?

The Domain Event handler components as introduced in Chapter 10 belong to the write side. Their purpose is to listen for event notifications and to execute consequential use case behavior that results in state changes. Except for the task board context, their implementations remain unchanged.

Shared code and directory structure

The implementation of CQRS requires to introduce three additional generic functionalities and to change the directory structure of all contexts. For Command and Query type definitions, the component `messageTypeFactory` is used. Also, the function `createMessageForwarder()` is added to construct categorized handlers for Commands, Queries and Domain Events. Furthermore, an indexed in-memory storage component is implemented. The context directory structure is adjusted to differentiate between a write side and a read side. This is done by introducing the two top-level directories “read-side” and “write-side” within each context. Both of them follow the same subdirectory pattern as previously used to categorize individual components according to their architectural layers. Furthermore, the top-level directory “domain” is added to each context for shared domain-related aspects, such as Domain Event definitions.

The following code shows the indexed in-memory storage component ([run code](#)):

Infrastructure code: Indexed in-memory storage

```
class InMemoryIndexedStorage {

    #indexes; #recordsById = new Map(); #indexMaps;

    constructor({indexes = []}) {
        this.#indexes = indexes;
        this.#indexMaps = Object.fromEntries(indexes.map(index => [index, new Map()]));
    }

    update(id, updates) {
        const oldRecord = this.#recordsById.get(id) || {};
        const newRecord = {...oldRecord, ...updates};
        this.#recordsById.set(id, newRecord);
        const indexesToUpdate = this.#indexes.filter(index => index in updates);
        for (const index of indexesToUpdate) {
            if (index in oldRecord) {
                const oldIndexIds = this.#indexMaps[index].get(`${oldRecord[index]}`);
                const idIndex = oldIndexIds.indexOf(id);
                if (idIndex > -1) oldIndexIds.splice(idIndex, 1);
            }
            const newIndexIds = this.#indexMaps[index].get(`${newRecord[index]}`) || [];
            this.#indexMaps[index].set(`${newRecord[index]}`, newIndexIds.concat(id));
        }
    }
}
```

```
load(id) {
    return this.#recordsById.get(id);
}

findByIndex(index, indexValue) {
    return (this.#indexMaps[index].get(indexValue) || [])
        .map(id => this.#recordsById.get(id));
}

}
```

The class `InMemoryIndexedStorage` enables to store records in-memory and to retrieve them by identifier and by custom indexes. Its constructor expects an index list and creates multiple `Map` objects to reference stored data by all required dimensions. Creating and updating a record is done with the command `update()`. The operation expects an identifier and a data structure. As first step, any existing record is retrieved. Then, this record is combined with the new data and the result is stored. Afterwards, all affected indexes are updated. For each one, the given identifier is unlinked from the old index collection and linked to the new one. The operation `load()` retrieves an item by identifier. Finally, the query `findByIndex()` returns all records that match a given index value.



Assume asynchronous interface

The following Application Layer components treat the indexed storage component as asynchronous, despite the fact the in-memory implementation works synchronously. This approach ensures that the component can be replaced with an asynchronous implementation without changing the consumer code.

Project context

The Application Services component of the project context is split into write and read concerns. All the write-related behavior is transformed into Commands and according Command Handlers. This incorporates the use cases for creating projects, updating their names, adding and removing team members, and updating member roles. In contrast, the read-related service for finding all projects of individual users is implemented as a Query functionality. The Domain Layer of the context is adapted in different ways. For one, specialized Domain Event types are introduced for all relevant state changes. Secondly, the Write Model implementation is extended to create the according event instances. For the

synchronization of Read Model data, the Application Layer is extended with a component that processes selected event notifications.

The first examples provide the definitions for all Commands and Queries:

Project context: Commands

```
const CreateProjectCommand = createMessageType('CreateProject', {name: 'string',
  projectId: 'string', ownerId: 'string', teamId: 'string', taskBoardId: 'string'});
const AddTeamMemberToTeamCommand = createMessageType('AddTeamMemberToTeam',
  {teamId: 'string', teamMemberId: 'string', userId: 'string', role: 'string'});
const RemoveTeamMemberFromTeamCommand = createMessageType(
  'RemoveTeamMemberFromTeam', {teamId: 'string', teamMemberId: 'string'});
const UpdateTeamMemberRoleCommand = createMessageType(
  'UpdateTeamMemberRole', {teamMemberId: 'string', role: 'string'});
const UpdateProjectNameCommand = createMessageType(
  'UpdateProjectName', {projectId: 'string', name: 'string'});
```

Project context: Queries

```
const FindProjectsByCollaboratingUserQuery = createMessageType(
  'FindProjectsByCollaboratingUser', {userId: 'string'});
const FindProjectsByOwnerQuery = createMessageType(
  'FindProjectsByOwner', {userId: 'string'});
```

The next code shows the full set of Domain Event types:

Project context: Domain Events

```
const ProjectCreatedEvent = createEventType('ProjectCreated', {projectId: 'string',
  name: 'string', ownerId: 'string', teamId: 'string', taskBoardId: 'string'});
const ProjectRenamedEvent = createEventType(
  'ProjectRenamed', {projectId: 'string', name: 'string'});
const TeamMemberCreatedEvent = createEventType(
  'TeamMemberCreated', {teamMemberId: 'string', userId: 'string', role: 'string'});
const TeamMemberAddedToTeamEvent = createEventType(
  'TeamMemberAddedToTeam', {teamId: 'string', teamMemberId: 'string'});
const TeamMemberRemovedFromTeamEvent = createEventType(
  'TeamMemberRemovedFromTeam', {teamId: 'string', teamMemberId: 'string'});
```

The following three examples illustrate the required changes to the Write Model implementation:

Project context: Project Entity name setter

```
set name(name) {
    verify('valid name', typeof name == 'string' && !!name);
    this.#name = name;
    this.#newDomainEvents.push(
        new ProjectRenamedEvent({projectId: this.id, name: this.name}));
}
```

Project context: Team Member constructor

```
constructor({id, userId, role, isExistingMember}) {
    verify('valid id', id != null);
    verify('valid user id', userId != null);
    Object.defineProperty(this, 'id', {value: id, writable: false});
    Object.defineProperty(this, 'userId', {value: userId, writable: false});
    this.role = role;
    if (!isExistingMember) this.#newDomainEvents.push(
        new TeamMemberCreatedEvent({teamMemberId: id, userId, role: role.name}));
}
```

Project context: Team add member function

```
addMember(teamMemberId) {
    verify('team member is new', !this.#teamMemberIds.includes(teamMemberId));
    this.#teamMemberIds.push(teamMemberId);
    this.#newDomainEvents.push(
        new TeamMemberAddedToTeamEvent({teamId: this.id, teamMemberId}));
}
```

The Command definitions represent all message types that are related to write use cases. For the read side, two specialized Query types are introduced. The Domain Event definitions extend the previously existing types `ProjectCreatedEvent` and `TeamMemberRemovedFromTeamEvent`. For the project Entity, the function `set name()` is adapted to create a “`ProjectRenamed`” event. The constructor of the `TeamMember` class is extended with a flag for reconstitution and the creation of a “`TeamMemberCreated`” event. Furthermore, the Entity type is complemented with the factory `TeamMemberFactory`. The operation `addMember()` of the `Team` component is refactored to create an instance of the `TeamMemberAddedToTeamEvent` type. Finally, all Repository components are reduced to their constructor. The previously introduced custom query operations are obsolete due to the introduction of a Read Model.

The next code implements a Read Model synchronization component:

Project context: Read Model synchronization

```
class ProjectReadModelSynchronization {

    #projectReadModelStorage; #teamMemberReadModelStorage; #eventBus;

    constructor({projectReadModelStorage, teamMemberReadModelStorage, eventBus}) {
        this.#projectReadModelStorage = projectReadModelStorage;
        this.#teamMemberReadModelStorage = teamMemberReadModelStorage;
        this.#eventBus = eventBus;
    }

    handleEvent = createMessageForwarder(this, {messageSuffix: 'Event'});

    activate() {
        const domainEventsToHandle = [ProjectCreatedEvent,
            ProjectRenamedEvent, TeamMemberCreatedEvent,
            TeamMemberAddedToTeamEvent, TeamMemberRemovedFromTeamEvent];
        domainEventsToHandle.forEach(Event =>
            this.#eventBus.subscribe(Event.type, event => this.handleEvent(event)));
    }

    async handleProjectCreatedEvent(event) {
        const {data: {projectId, name, ownerId, teamId, taskBoardId}} = event;
        const updates = {id: projectId, name, ownerId, teamId, taskBoardId};
        await this.#projectReadModelStorage.update(projectId, updates);
    }

    async handleProjectRenamedEvent({data: {projectId, name}}) {
        await this.#projectReadModelStorage.update(projectId, {name});
    }

    async handleTeamMemberCreatedEvent({data: {teamMemberId: id, userId}}) {
        await this.#teamMemberReadModelStorage.update(id, {id, userId});
    }

    async handleTeamMemberAddedToTeamEvent({data: {teamMemberId: id, teamId}}) {
        await this.#teamMemberReadModelStorage.update(id, {id, teamId});
    }

    async handleTeamMemberRemovedFromTeamEvent({data: {teamMemberId: id}}) {
        await this.#teamMemberReadModelStorage.update(id, {teamId: null});
    }
}
```

```
}
```

This is followed by an according Query Handlers class:

Project context: Query Handlers

```
class ProjectQueryHandlers {

    #projectReadModelStorage; #teamMemberReadModelStorage;

    constructor({projectReadModelStorage, teamMemberReadModelStorage}) {
        this.#projectReadModelStorage = projectReadModelStorage;
        this.#teamMemberReadModelStorage = teamMemberReadModelStorage;
    }

    handleQuery = createMessageForwarder(this, {messageSuffix: 'Query'});

    async handleFindProjectsByOwnerQuery({data: {userId}}) {
        return await this.#projectReadModelStorage.findIndex('ownerId', userId);
    }

    async handleFindProjectsByCollaboratingUserQuery({data: {userId}}) {
        const teamMembers =
            await this.#teamMemberReadModelStorage.findIndex('userId', userId);
        const teamIds = teamMembers.map(teamMember => teamMember.teamId);
        const projectsByTeamMemberId = await Promise.all(
            teamIds.map(id => this.#projectReadModelStorage.findIndex('teamId', id)));
        return projectsByTeamMemberId.flat(1);
    }
}
```

The component `ProjectReadModelSynchronization` creates and updates Read Models for projects and team members. Its constructor expects two storage components and an Event Bus instance. Creating a unified subscriber is done with the message forwarder factory. The operations `handleTeamMemberCreatedEvent()` and `handleTeamMemberAddedToTeamEvent()` work independently of event ordering. This is required because the respective events have no guaranteed order, as they belong to different transactional consistency boundaries. The component `ProjectQueryHandlers` handles Queries by responding with Read Model results. Its constructor also expects two storage components. The use case of finding all projects for collaborating users consists of multiple steps. First, all associated team members are

determined. For each of them, all projects of the respective team are loaded. Finally, a flattened list is returned as result.

The usage of the full project context implementation with CQRS can be seen in the next example ([run code](#)):

Project context: Overall usage

```
const projectCommandHandlers = new ProjectCommandHandlers({teamRepository,
  teamMemberRepository, projectRepository, teamMemberFactory, projectFactory});
const projectDomainEventHandlers = new ProjectDomainEventHandlers(
  {teamRepository, eventBus});
projectDomainEventHandlers.activate();
const projectReadModelStorage =
  new IndexedStorage({indexes: ['ownerId', 'teamId']});
const teamMemberReadModelStorage = new IndexedStorage({indexes: ['userId']});
const projectReadModelSynchronization = new ProjectReadModelSynchronization(
  {projectReadModelStorage, teamMemberReadModelStorage, eventBus});
projectReadModelSynchronization.activate();
const projectQueryHandlers = new ProjectQueryHandlers(
  {projectReadModelStorage, teamMemberReadModelStorage});

const teamId = generateId(), teamMemberId = generateId(), userId = generateId();
const projectId = generateId(), taskBoardId = generateId();

projectCommandHandlers.handleCommand(new CreateProjectCommand(
  {data: {projectId, name: 'Test Project', ownerId: userId, teamId, taskBoardId}}));
await timeout(125);
projectCommandHandlers.handleCommand(new AddTeamMemberToTeamCommand(
  {data: {teamId, teamMemberId, userId, role: 'developer'}}));
projectCommandHandlers.handleCommand(new UpdateProjectNameCommand(
  {data: {projectId, name: 'Test Project v2'}}));
await timeout(125);
console.log(await projectQueryHandlers.handleQuery(
  new FindProjectsByOwnerQuery({data: {userId}})));
console.log(await projectQueryHandlers.handleQuery(
  new FindProjectsByCollaboratingUserQuery({data: {userId}})));
```

User context

At first glance, the Application Services for the user context exclusively consist of write-related behavior. Creating a user, performing a login and updating existing fields are all

Command operations. However, there are two implicit read concerns. One is that the login service queries the user Repository by e-mail addresses. This aspect is extracted into a Query functionality for retrieving individual user data. For that, the Domain Model implementation is extended with new Domain Event types and Entities are adapted accordingly. Creating and updating the required Read Model structure is done by a synchronization component. The second read aspect is the e-mail registry, which contains eventually consistent data. Strictly speaking, this part is already a Read Model, despite the fact it exists within the write side.



Read Models in the write side

Read Models are eventually consistent derived data structures. While they are typically used to maintain information within a read side, they can also exist in other architectural areas. Creating a specialized Read Model within a write side can help to enforce a rule across transactional boundaries.

The following examples show all Command and Query types:

User context: Commands

```
const CreateUserCommand = createMessageType('CreateUser', {userId: 'string',
  username: 'string', emailAddress: 'string', password: 'string', role: 'string'});
const LoginUserCommand = createMessageType(
  'LoginUser', {userId: 'string', password: 'string'});
const UpdateUsernameCommand = createMessageType(
  'UpdateUsername', {userId: 'string', username: 'string'});
const UpdateUserPasswordCommand = createMessageType(
  'UpdateUserPassword', {userId: 'string', password: 'string'});
const UpdateUserEmailAddressCommand = createMessageType(
  'UpdateUserEmailAddress', {userId: 'string', emailAddress: 'string'});
const UpdateUserRoleCommand = createMessageType(
  'UpdateUserRole', {userId: 'string', role: 'string'});
```

User context: Queries

```
const FindUserByEmailAddressQuery = createMessageType(
  'FindUserByEmailAddress', {emailAddress: 'string'});
```

The next example provides the Domain Event definitions:

User context: Domain Events

```
const UserCreatedEvent = createEventType('UserCreated',
  {userId: 'string', username: 'string', emailAddress: 'string', role: 'string'});
const UsernameChangedEvent = createEventType(
  'UsernameChanged', {userId: 'string', username: 'string'});
const UserEmailAddressChangedEvent = createEventType(
  'UserEmailAddressChanged', {userId: 'string', emailAddress: 'string'});
const UserRoleChangedEvent = createEventType(
  'UserRoleChanged', {userId: 'string', role: 'string'});
```

The changes to the user Entity can be seen in the next examples:

User context: User constructor

```
constructor({id, username, emailAddress, password, role,
  emailAvailability, isExistingUser}) {
  verify('valid id', id != null);
  verify('valid username', typeof username == 'string' && !!username);
  verify('valid e-mail address', emailAddress instanceof EmailAddress);
  verify('unused e-mail',
    isExistingUser || emailAvailability.isEmailAvailable(emailAddress));
  verify('valid role', role.constructor === Role);
  Object.defineProperty(this, 'id', {value: id, writable: false});
  this.#emailAvailability = emailAvailability;
  this.#username = username;
  this.#emailAddress = emailAddress;
  this.#role = role;
  this.password = password;
  if (!isExistingUser) this.#newDomainEvents.push(new UserCreatedEvent(
    {userId: id, username, emailAddress: emailAddress.value, role: role.name}));
```

User context: User name setter

```
set username(username) {
    verify('valid username', typeof username == 'string' && !username);
    this.#username = username;
    this.#newDomainEvents.push(new UsernameChangedEvent(
        {userId: this.id, username}));
}
```

User context: User role setter

```
set role(role) {
    verify('valid role', role.constructor === Role);
    this.#role = role;
    this.#newDomainEvents.push(
        new UserRoleChangedEvent({userId: this.id, role: role.name}));
}
```

The Command types define all the messages related to write use cases. For the functionality of retrieving a user profile, the Query type `FindUserByEmailAddressQuery` is introduced. The updated Domain Event definitions contain a total of four types that represent relevant state changes. As replacement for the previously existing `UserEmailAddressAssignedEvent`, the two more explicit types “`UserCreated`” and “`UserEmailAddressChanged`” are added. The user Entity is extended to create according events in its constructor and in both the operations `setName()` and `set role()`. As there are no changes to arguments or attributes, the user factory remains unchanged. In contrast, the Repository component is reduced to its constructor by removing the custom query `findUserByEmailAddress()`. The e-mail registry Read Model component requires no refactoring.

The following code provides the implementation of a Read Model synchronization component:

User context: Read Model synchronization

```
class UserReadModelSynchronization {

    #userReadModelStorage; #eventBus;

    constructor({userReadModelStorage, eventBus}) {
        this.#userReadModelStorage = userReadModelStorage;
        this.#eventBus = eventBus;
    }

    activate() {
        const domainEventsToHandle = [UserCreatedEvent, UsernameChangedEvent,
            UserEmailAddressChangedEvent, UserRoleChangedEvent];
        domainEventsToHandle.forEach(Event =>
            this.#eventBus.subscribe(Event.type, event => this.handleEvent(event)));
    }

    handleEvent = createMessageForwarder(this, {messageSuffix: 'Event'});

    async handleUserCreatedEvent({data: {userId: id, username, emailAddress, role}}) {
        await this.#userReadModelStorage.update(id, {id, username, emailAddress, role});
    }

    async handleUsernameChangedEvent({data: {userId, username}}) {
        await this.#userReadModelStorage.update(userId, {username});
    }

    / * .. handleUserEmailAddressChangedEvent(), handleUserRoleChangedEvent() .. */

}
```

The next example shows the Query Handlers component for the user context:

User context: Query Handlers

```
class UserQueryHandlers {  
  
    #userReadModelStorage;  
  
    constructor({userReadModelStorage}) {  
        this.#userReadModelStorage = userReadModelStorage;  
    }  
  
    handleQuery = createMessageForwarder(this, {messageSuffix: 'Query'});  
  
    async handleFindUserByEmailAddressQuery({data: {emailAddress}}) {  
        return await this.#userReadModelStorage.findByIndex(  
            'emailAddress', emailAddress);  
    }  
}
```

The class `UserReadModelSynchronization` creates and updates user-related Read Model data. Its constructor expects a storage component and an Event Bus instance. Again, a unified event subscriber mechanism is created by executing the operation `createMessageForwarder()`. While the handler function `handleUserCreatedEvent()` saves a new Read Model entry, all the other operations update attributes of existing items. The component `UserQueryHandlers` handles the Query type “`FindUserByEmailAddress`” and returns the corresponding user Read Model data. Note that in exceptional cases, this service can yield multiple entries for the same e-mail address. This is because ensuring the uniqueness of e-mail addresses is not a transactional process on the write side. By returning all matching items, the responsibility of what action to take is shifted to the issuer of the Query.



Memory consumption of a Read Model

Storing Read Models exclusively in memory can work well, even for moderately large amounts of data. As example, assume that the e-mail address of a user has a maximum length of 200. In JavaScript, each string character consumes 2 bytes. Even with ten million users, the data for the e-mail registry would require less than 4 gigabytes of memory.

The last example of this subsection provides an example usage of the user context ([run code](#)):

User context: Overall usage

```
const userCommandHandlers = new UserCommandHandlers(
  {userRepository, userFactory, hashPassword: createMd5Hash});
const userDomainEventHandlers = new UserDomainEventHandlers(
  {emailRegistry, eventBus});
userDomainEventHandlers.activate();

const userReadModelStorage = new IndexedStorage({indexes: ['emailAddress']});
const userProfilesSynchronization = new ReadModelSynchronization(
  {userReadModelStorage, eventBus});
const userQueryHandlers = new UserQueryHandlers({userReadModelStorage});
userProfilesSynchronization.activate();

const randomId = generateId();
const emailAddress1 = `johnathan${randomId}@example.com`;
const emailAddress2 = `john${randomId}@example.com`;
const userId1 = generateId(), userId2 = generateId();

(await userRepository.loadAll()).forEach(user =>
  emailRegistry.setUserEmailAddress(user.id, user.emailAddress));
const commandsToExecute = [
  new CreateUserCommand({data: {userId: userId1, username: 'johnathan',
    emailAddress: emailAddress1, password: 'pw1', role: 'user'}}),
  new UpdateUserEmailAddressCommand(
    {data: {userId: userId1, emailAddress: emailAddress2}}),
];
for (const command of commandsToExecute)
  await userCommandHandlers.handleCommand(command);
await timeout(125);
const result = await userQueryHandlers.handleQuery(
  new FindUserByEmailAddressQuery({data: {emailAddress: emailAddress2}}));
console.log(result);
await userCommandHandlers.handleCommand(new CreateUserCommand({data:
  {userId: userId2, username: 'john',
    emailAddress: emailAddress2, password: 'pw1', role: 'user'}}));
```

Task board context

For the task board context, the majority of Application Services is concerned with state changes. The use cases consist of adding new tasks to a board, updating existing items

and removing tasks from a board. The only read functionality is the retrieval of tasks on a board, which is extracted into a Query. The Domain Layer is updated with additional Domain Events and according Entity modifications. For maintaining Read Model data, a synchronization component is introduced. Apart from the actual use cases, there is another read aspect inside a specific event handler. The un-assignment of tasks for removed team members depends on the Repository query `findTasksByAssigneeId()`. Similar to the e-mail registry, this functionality can be replaced with a specialized Read Model within the write side.

The first examples provide the types for all Commands, Queries and Domain Events:

Task board context: Commands

```
const AddNewTaskToTaskBoardCommand = createMessageType('AddNewTaskToTaskBoard', {
  taskId: 'string', taskBoardId: 'string', title: 'string', description: 'string',
  status: ['string', 'undefined'], assigneeId: ['string', 'undefined'],
});
const UpdateTaskTitleCommand = createMessageType(
  'UpdateTaskTitle', {taskId: 'string', title: 'string'});
const UpdateTaskDescriptionCommand = createMessageType(
  'UpdateTaskDescription', {taskId: 'string', description: 'string'});
const UpdateTaskStatusCommand = createMessageType(
  'UpdateTaskStatus', {taskId: 'string', status: 'string'});
const UpdateTaskAssigneeCommand = createMessageType(
  'UpdateTaskAssignee', {taskId: 'string', assigneeId: 'string'});
const RemoveTaskFromTaskBoardCommand = createMessageType(
  'RemoveTaskFromTaskBoard', {taskBoardId: 'string', taskId: 'string'});
```

Task board context: Queries

```
const FindTasksOnTaskBoardQuery = createMessageType(
  'FindTasksOnTaskBoard', {taskBoardId: 'string'});
```

Task Board context: Domain Events

```
const TaskCreatedEvent = createEventType('TaskCreated',
  {taskId: 'string', title: 'string', description: 'string',
   status: ['string', 'undefined'], assigneeId: ['string', 'undefined']}));
const TaskTitleChangedEvent = createEventType(
  'TaskTitleChanged', {taskId: 'string', title: 'string'});
const TaskDescriptionChangedEvent = createEventType(
  'TaskDescriptionChanged', {taskId: 'string', description: 'string'});
const TaskAssigneeChangedEvent = createEventType(
  'TaskAssigneeChanged', {taskId: 'string', assigneeId: ['string', 'undefined']}));
const TaskStatusChangedEvent = createEventType(
  'TaskStatusChanged', {taskId: 'string', status: 'string'});
const TaskAddedToTaskBoardEvent = createEventType(
  'TaskAddedToTaskBoard', {taskBoardId: 'string', taskId: 'string'});
const TaskRemovedFromTaskBoardEvent = createEventType(
  'TaskRemovedFromTaskBoard', {taskBoardId: 'string', taskId: 'string'})
```

The next code snippets show the required changes to the Task Entity class:

Task Board context: Task constructor

```
constructor(
  {id, title, description = '', status = 'todo', assigneeId, isExistingTask}) {
  verify('valid id', id != null);
  verify('valid title', typeof title == 'string' && !!title);
  verify('valid status', validStatus.includes(status));
  verify('active task assignee', status !== 'in progress' || assigneeId);
  Object.defineProperty(this, 'id', {value: id, writable: false});
  this.#title = title;
  this.#description = description;
  this.#status = status;
  this.#assigneeId = assigneeId;
  if (!isExistingTask) this.#newDomainEvents.push(
    new TaskCreatedEvent({taskId: id, title, description, status, assigneeId}));
}
```

Task Board context: Task title setter

```
set title(title) {
    verify('valid title', typeof title == 'string' && !title);
    this.#title = title;
    this.#newDomainEvents.push(new TaskTitleChangedEvent({taskId: this.id, title}));
}
```

Task Board context: Task description setter

```
set description(description) {
    this.#description = description;
    this.#newDomainEvents.push(
        new TaskDescriptionChangedEvent({taskId: this.id, description}));
}
```

Task Board context: Task assignee setter

```
set assigneeId(assigneeId) {
    verify('active task assignee', this.status !== 'in progress' || assigneeId);
    this.#assigneeId = assigneeId;
    this.#newDomainEvents.push(
        new TaskAssigneeChangedEvent({taskId: this.id, assigneeId}));
}
```

The following example provides a reworked version of the Task Board Entity type:

Task Board context: Task Board

```
class TaskBoard {

    id; #taskIds; #newDomainEvents = [];

    constructor({id, initialTaskIds = []}) {
        verify('valid id', id != null);
        Object.defineProperty(this, 'id', {value: id, writable: false});
        this.#taskIds = initialTaskIds;
    }

    addTask(taskId) {
        verify('valid task id', taskId != null);
        this.#taskIds.push(taskId);
    }
}
```

```

    this.#newDomainEvents.push(
      new TaskAddedToTaskBoardEvent({taskBoardId: this.id, taskId}));
  }

removeTask(taskId) {
  const index = this.#taskIds.indexOf(taskId);
  verify('task is on board', index > -1);
  this.#taskIds.splice(index, 1);
  this.#newDomainEvents.push(
    new TaskRemovedFromTaskBoardEvent({taskBoardId: this.id, taskId}));
}

get taskIds() { return this.#taskIds.slice(); }

get newDomainEvents() { return this.#newDomainEvents.slice(); }

}

```

The next code shows an extended variant of the Domain Event handlers component:

Task Board context: Domain Event Handlers

```

class TaskBoardDomainEventHandlers {

  constructor({taskRepository, taskBoardRepository,
    taskAssigneeReadModelStorage, eventBus}) {
    this.activate = () => {
      eventBus.subscribe('ProjectCreated', () => {/* .. */});
      eventBus.subscribe(TaskCreatedEvent.type, ({data: {taskId, assigneeId}}) => {
        taskAssigneeReadModelStorage.update(taskId, {id: taskId, assigneeId});
      });
      eventBus.subscribe(TaskAssigneeChangedEvent.type, event => {
        const {data: {taskId, assigneeId}} = event;
        taskAssigneeReadModelStorage.update(taskId, {id: taskId, assigneeId});
      });
      eventBus.subscribe('TeamMemberRemovedFromTeam', async ({data}) => {
        const tasks =
          taskAssigneeReadModelStorage.findIndex('assigneeId', data.teamMemberId);
        await Promise.all(taskAssignees.map(async ({id}) => {
          const task = await taskRepository.load(id);
          if (task.status === 'in progress') task.status = 'todo';
          task.assigneeId = undefined;
          await taskRepository.save(task);
        }));
      });
    }
  }
}

```

```
    });
};

}

}
```

The Command types define the messages for all write-related use cases. Finding tasks on a board is represented by the Query type “FindTasksOnTaskBoard”. The Domain Event types define all relevant state changes that can occur within the Domain Model. For the task Entity, the constructor and the setters for title, description and assignee are extended to create according events. The task board Entity is completely refactored. Instead of working with specialized Value Objects, the component exclusively references task identifiers. The Task class accepts an additional constructor argument for Entity reconstitution. For this reason, an according factory is implemented. The custom query `findTasksByAssigneeId()` is removed from the task Repository. As replacement, the component `TaskBoardDomainEventHandlers` is adapted to maintain a Read Model of task assignees inside an `InMemoryIndexedStorage` instance.

The next code provides a synchronization component for task Read Models:

Task board context: Read Model synchronization

```
class TaskBoardReadModelSynchronization {

    #taskReadModelStorage; #eventBus;

    constructor({taskReadModelStorage, eventBus}) {
        this.#taskReadModelStorage = taskReadModelStorage;
        this.#eventBus = eventBus;
    }

    handleEvent = createMessageForwarder(this, {messageSuffix: 'Event'});

    activate() {
        const domainEventsToHandle = [TaskCreatedEvent, TaskAddedToTaskBoardEvent,
            TaskRemovedFromTaskBoardEvent, TaskDescriptionChangedEvent,
            TaskTitleChangedEvent, TaskAssigneeChangedEvent, TaskStatusChangedEvent];
        domainEventsToHandle.forEach(Event =>
            this.#eventBus.subscribe(Event.type, event => this.handleEvent(event)));
    }

    async handleTaskCreatedEvent({data}) {
```

```

    const {taskId, title, description, status, assigneeId} = data;
    const updates = {id: taskId, title, description, status, assigneeId};
    await this.#taskReadModelStorage.update(taskId, updates);
}

async handleTaskAddedToTaskBoardEvent({data: {taskId, taskBoardId}}) {
    await this.#taskReadModelStorage.update(taskId, {id: taskId, taskBoardId});
}

/* .. handleTaskRemovedFromTaskBoardEvent() .. */

async handleTaskTitleChangedEvent({data: {taskId, title}}) {
    await this.#taskReadModelStorage.update(taskId, {title});
}

/* .. handleTaskDescriptionChangedEvent() .. */
/* .. handleTaskAssigneeChangedEvent() .. */
/* .. handleTaskStatusChangedEvent() .. */

}

```

This is followed by the implementation of the Query Handlers class:

Task Board context: Query Handlers

```

class TaskBoardQueryHandlers {

    #taskReadModelStorage;

    constructor({taskReadModelStorage}) {
        this.#taskReadModelStorage = taskReadModelStorage;
    }

    handleQuery = createMessageForwarder(this, {messageSuffix: 'Query'});

    async handleFindTasksOnTaskBoardQuery({data: {taskBoardId}}) {
        return await this.#taskReadModelStorage.findIndex('taskBoardId', taskBoardId);
    }

}

```

The class `TaskBoardReadModelSynchronization` maintains Read Model data for tasks together with their associated task board identifier. Similar to the other synchronization classes,

the constructor expects a storage component and an Event Bus instance. Also, a message forwarding mechanism is created with the factory `createMessageForwarder()`. The two operations `handleTaskCreatedEvent()` and `handleTaskAddedToTaskBoardEvent()` work independently of event ordering, as they process messages from different transactional boundaries. All the other handler operations are responsible for updating Read Model data in response to changes of individual task attributes. The class `TaskBoardQueryHandlers` implements the read-related use case behavior. For the Query type “`FindTasksOnTaskBoard`”, the responsible handler function retrieves all affected task entries. This is done by querying the indexed storage component using the task board index.

The final example of this chapter shows an exemplary usage of the task board context ([run code](#)):

Task board context: Overall usage

```
const taskAssigneeReadModelStorage =
  new InMemoryIndexedStorage({indexes: ['assigneeId']});
const taskBoardCommandHandlers = new TaskBoardCommandHandlers(
  {taskRepository, taskBoardRepository, taskFactory});
const taskBoardDomainEventHandlers = new TaskBoardDomainEventHandlers(
  {taskRepository, taskBoardRepository, taskAssigneeReadModelStorage, eventBus});
taskBoardDomainEventHandlers.activate();

const taskReadModelStorage = new InMemoryIndexedStorage({indexes: ['taskBoardId']});
const readModelSynchronization = new ReadModelSynchronization(
  {taskReadModelStorage, eventBus});
const taskBoardQueryHandlers = new TaskBoardQueryHandlers({taskReadModelStorage});
readModelSynchronization.activate();

const taskBoardId = generateId(), taskId = generateId(), assigneeId = generateId();

await eventBus.publish({type: 'ProjectCreated', data: {taskBoardId}});
await taskBoardCommandHandlers.handleCommand(new AddNewTaskToTaskBoardCommand(
  {data: {taskId, taskBoardId, title: 'tests', description: 'write unit tests'}}));
await taskBoardCommandHandlers.handleCommand(
  new UpdateTaskAssigneeCommand({data: {taskId, assigneeId}}));
await taskBoardCommandHandlers.handleCommand(
  new UpdateTaskStatusCommand({data: {taskId, status: 'in progress'}}));
await timeout(125);
console.log(await taskBoardQueryHandlers.handleQuery(
  new FindTasksOnTaskBoardQuery({data: {taskBoardId}})));
eventBus.publish(new TeamMemberRemovedFromTeamEvent(
  {teamId: generateId(), teamMemberId: assigneeId}));
await timeout(125);
```

```
console.log(await taskBoardQueryHandlers.handleQuery(  
    new FindTasksOnTaskBoardQuery({data: {taskBoardId}})));
```

The Sample Application implementation with CQRS cleanly separates the write side and the read side of each individual context. Also, it eliminates performance problems due to expensive Repository queries through optimized Read Model structures. However, for a productive use, the in-memory storage component must be complemented with a rebuild mechanism. The next step is to transform the Domain Models and their implementations into an event-based state representation.

Chapter 12: Event Sourcing

Event Sourcing is an architectural pattern where state is represented as a sequence of change events. The events are treated as immutable and get persisted in an append-only log. Any current data is computed on demand and generally considered transient. Capturing state as a set of transitions enables to emphasize intentions and behavior instead of focusing primarily on data structures. According to [Young], another benefit is that “time becomes an important factor” of a system. Yet, the most distinct advantage is that historical event logs allow to derive specialized data for read-related scenarios, even retroactively. Event Sourcing can only be applied adequately together with some form of CQRS. Independent of this constraint, the pattern is commonly used in combination with both Domain Events and Event-Driven Architecture.

Architectural overview

Event Sourcing models state as a series of transitioning events. As with CQRS, the pattern typically targets subordinate areas or specific contexts, but rarely a complete software. Even more, individual parts of a system should not rely on the fact that other areas apply this pattern. Instead, it should be treated as an internal implementation detail. However, it is acceptable to provide query functionalities that return historical or event-related information in a controlled manner. Other than with CQRS, Event Sourcing only affects certain architectural parts. This primarily includes the Write Model implementation, its associated infrastructural functionalities and the synchronization mechanism for Read Model data. In contrast, most parts of the read side, its storage components and the User Interface Layer largely remain unaffected.



Relation between CQRS and Event Sourcing

CQRS was originally meant to be a stepping stone when working towards Event Sourcing. However, the pattern can also be applied independently and its core idea is even found in other concepts. In contrast, Event Sourcing primarily makes sense when a software also separates its write side and read side.

Overall flow

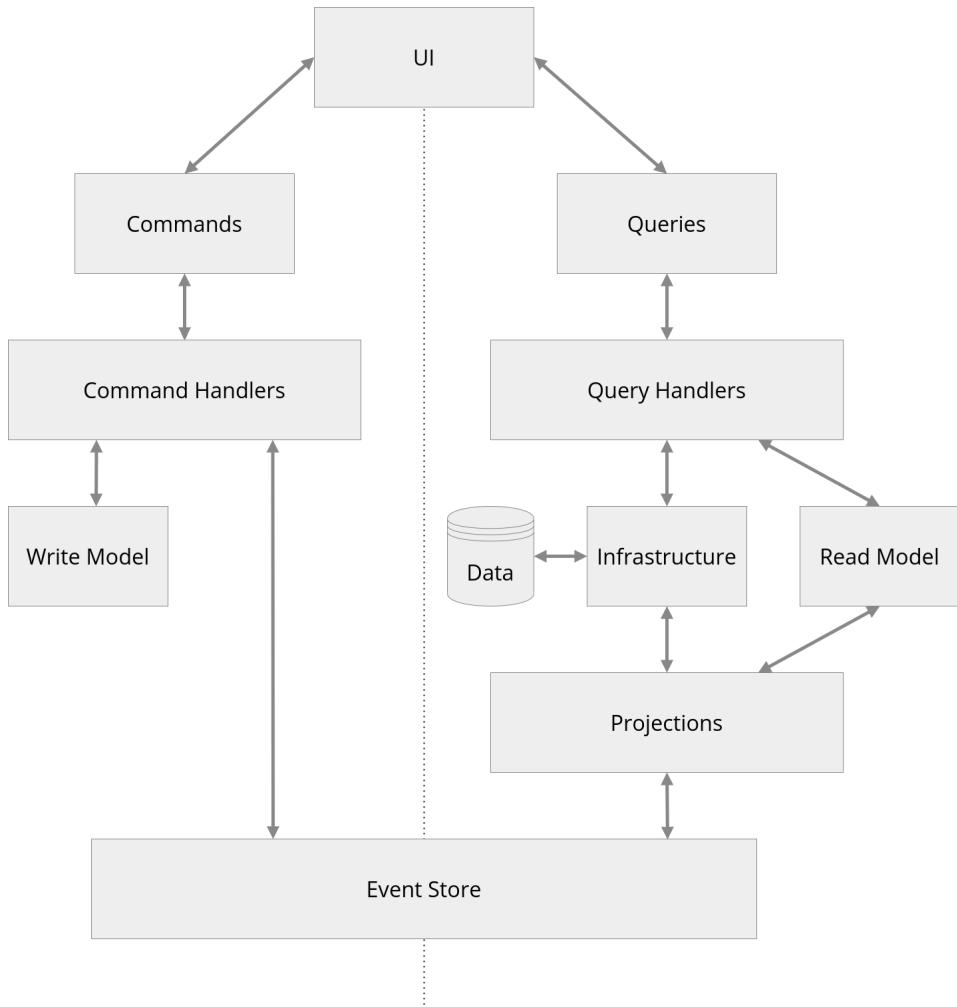


Figure 12.1: Event Sourcing

The write side retrieves events from the Event Store, rebuilds state representations for Write Model components and executes target behavior. All new events are appended to their according stream inside the store. For DDD-based software, the event streams are typically separated by Aggregates. After persisting, all interested subscribers are informed about the state changes. The read side operates Projection components, which subscribe at the Event Store to update Read Model structures. Upon notification, they access data from

their associated storage, optionally construct Read Model components and perform necessary updates. For specialized use cases, the read side may even expose direct event stream access. In terms of CQRS, the Event Store is both the persistence mechanism of the write side and part of the synchronization mechanism.

Event anatomy

The recommendations for Domain Events with regard to types, structure and content largely also apply to Event Sourcing. Every event should have an expressive and unambiguous type name that adequately describes the enclosed change. The term should be a combination of the affected subject and the executed action in past tense. For each event, the contents should consist of both data and metadata, where individual fields are represented as simple attributes. The data part should include the type name, the identity of the affected subject and all changed values. Other than with Domain Events, there should be no derived or extraneous data. The metadata section should contain generic information independent of event types. When using Event Sourcing, this part often includes additional persistence-related fields.

Relation to Domain Events

While Domain Events and Event Sourcing events share similarities, they are conceptually not the same. Domain Events represent meaningful Domain occurrences and can be distributed for both internal and external consumption. They may contain redundant and additional data to satisfy consumer needs. Event Sourcing captures state changes as events and uses them as persistence records. These items may be ill-suited for an outbound distribution. They are rather fine-grained, should only contain changed data and may have a local meaning. Exposing them directly risks to couple external functionalities to internal details. Therefore, there should be a dedicated mechanism that translates them to Domain Events. At a minimum, this functionality filters out items that are considered private. Furthermore, it can perform domain-specific transformations, such as data enrichment.



Every event is a Domain Event

In fact, all events can be considered Domain Events. However, they may differ in their scope of validity. Private events are only meaningful inside their enclosing context. In contrast, public events are part of a larger conceptual boundary. This book only labels items that are made available for external consumption as Domain Events.

Example: Video platform channel subscription

Consider implementing the Event Sourcing events and the Domain Event creation mechanism for channel subscriptions of a video platform. The goal is to be able to inform channel owners whenever a subscription is added or removed. Overall, the possible state transitions for an individual subscription consist of adding it, updating its notification settings and removing it. Both the addition and the removal must be treated as public Domain Events in order to notify channel owners. In contrast, the notification settings update is considered a private detail. Even though this occurrence may also be useful for external consumption, there is initially no need for exposing it. The mechanism for yielding Domain Events should only forward the public event types and filter out all private items.

The following code provides the event definitions together with an operation for yielding Domain Events ([run code](#)):

Video platform: Channel subscription events

```
const SubscriptionAddedEvent = createEventType('SubscriptionAdded',
  {subscriptionId: 'string', userId: 'string', channelId: 'string'});

const SubscriptionNotificationsSetEvent = createEventType(
  'SubscriptionNotificationsSet',
  {subscriptionId: 'string', notifications: 'boolean'});

const SubscriptionRemovedEvent = createEventType(
  'SubscriptionRemoved', {subscriptionId: 'string'});

const publicEventTypes = [SubscriptionAddedEvent, SubscriptionRemovedEvent];
const yieldDomainEvents = event => publicEventTypes.some(
  Event => Event.type === event.type) ? [event] : [];

const subscriptionId = generateId();
const userId = generateId();
const channelId = generateId();

const events = [
  new SubscriptionAddedEvent({subscriptionId, userId, channelId}),
  new SubscriptionNotificationsSetEvent({subscriptionId, notifications: true}),
  new SubscriptionRemovedEvent({subscriptionId}),
];

const domainEvents = events.flatMap(yieldDomainEvents);
```

The code starts with creating the event types “SubscriptionAdded”, “SubscriptionNotificationsSet” and “SubscriptionRemoved”. Next, the array `publicEventTypes` is initialized with all public event types. Then, the function `yieldDomainEvents()` is implemented, which expects an event as argument and returns a list of Domain Events. For each addition or removal of a subscription, it returns the original item. For all other types, an empty array is returned. The actual usage code starts with defining identifiers for a subscription, a user and a channel. Afterwards, three events are instantiated, one for each defined type. The items are passed individually to the function `yieldDomainEvents()` and the combined return value is logged to the console. Executing the code demonstrates that the settings update does not appear as Domain Event.



Specialized Domain Event types

As alternative to the example implementation, consider a separate Domain Event type to represent a channel subscription count change. The data could consist of both the previous and the new subscriber count. For each subscription addition and removal, the event type would be instantiated and forwarded. The denormalized data would free consumers from manually determining the current subscription count.

Event-sourced Write Model

The Write Model implementation for a software based on Event Sourcing must exhibit specific behavior. For one, each Aggregate type must be able to build its required state representation from a list of events. Secondly, all behavioral operations must not directly mutate state, but produce events that represent a respective change. As optional side effect, the information may be used to synchronize its state representation. Finally, all new events must be exposed to the outside for persistence purposes. There are multiple advantages with event-sourced Write Models. According to [Young], modeling in events “forces you to think about behavior”. Also, state representations can be structured freely without accommodations for persistence. One potential downside is the additional complexity of an event-sourced design and the resulting implementation.

Implementation styles

There are mainly two different styles for the implementation of an event-sourced Write Model. One option is to apply the object-oriented paradigm. The second possibility is to

utilize selected concepts of Functional Programming. With regard to the Domain Layer of a software, these styles mostly differ in code design and syntax. However, they can further have an effect on non-functional software characteristics such as scalability. Still, there is no universal argument for either of the approaches. As with other aspects of software development, it heavily depends on the respective scenario. The following subsections explain, illustrate and compare the two styles as well as one additional variation. This is preceded by the introduction of an example topic, which is used for the implementation of each approach.

Example: Support ticket system

Consider implementing an event-sourced Write Model for a support ticket system. The goal is to provide a communication platform for issues. The use cases consist of opening a ticket, adding a comment and closing a ticket. Opening a new ticket requires to provide an issuer identity, a title and a description. While the title is constrained to 50 characters, the description limit is set to 500. Any number of comments can be added to a ticket as means of communication between the issuer and the support user. Closing a ticket indicates that no further work is planned, either because the issue is solved or became invalid. One indirect requirement is that adding a comment to a closed ticket causes it automatically to be re-opened.

The following code provides the definitions for the events of the Domain Model:

Support ticket system: Event definitions

```
const SupportTicketOpenedEvent = createEventType('SupportTicketOpened',
  {ticketId: 'string', authorId: 'string', title: 'string', description: 'string'});

const SupportTicketCommentedEvent = createEventType('SupportTicketCommented',
  {ticketId: 'string', authorId: 'string', message: 'string'});

const SupportTicketClosedEvent = createEventType(
  'SupportTicketClosed', {ticketId: 'string', resolution: 'string'});

const SupportTicketReOpenedEvent = createEventType(
  'SupportTicketReOpened', {ticketId: 'string'});
```

Object-oriented approach

The implementation of an event-sourced Write Model using an object-oriented approach is seemingly the more common variant. One possible reason is the overall popularity of

the programming paradigm. The idea is to combine both behavior and state into a single class. Typically, the constructor expects a list of events to build an initial state representation. Executing a behavioral function causes to add the resulting events to an internal collection. As a side effect, the associated state representation is updated. All new events must be accessed manually on the outside. One advantage of this approach is that the continuous state update allows the execution of multiple consecutive actions. One downside is the need to maintain an internal event collection and to provide an according retrieval mechanism.

The following code shows an object-oriented implementation of the Support Ticket Aggregate type:

Support ticket system: Object-oriented Support Ticket

```
class SupportTicket {  
  
    #ticketId; #isOpen; #newEvents = [];  
  
    constructor(events = []) {  
        events.forEach(event => this.#applyEvent(event));  
    }  
  
    #applyEvent({type, data}) {  
        if (type === SupportTicketOpenedEvent.type) {  
            this.#ticketId = data.ticketId;  
            this.#isOpen = true;  
        }  
        if (type === SupportTicketClosedEvent.type) this.#isOpen = false;  
        if (type === SupportTicketReOpenedEvent.type) this.#isOpen = true;  
    }  
  
    open({id: ticketId, authorId, title, description}) {  
        if (title.length > 50) throw new Error('title length exceeded');  
        if (description.length > 500) throw new Error('description length exceeded');  
        this.#applyNewEvent(  
            new SupportTicketOpenedEvent({ticketId, authorId, title, description}));  
    }  
  
    comment({authorId, message = ''}) {  
        if (!this.#isOpen) this.#applyNewEvent(  
            new SupportTicketReOpenedEvent({ticketId: this.#ticketId}));  
        this.#applyNewEvent(new SupportTicketCommentedEvent(  
            {ticketId: this.#ticketId, authorId, message}));  
    }  
}
```

```
close({resolution}) {
    if (!['solved', 'closed'].includes(resolution))
        throw new Error('invalid resolution');
    this.#applyNewEvent(
        new SupportTicketClosedEvent({ticketId: this.#ticketId, resolution}));
}

#applyNewEvent(event) {
    this.#newEvents.push(event);
    this.#applyEvent(event);
}

get newEvents() { return this.#newEvents.slice(); }

}
```

The class `SupportTicket` expresses the support ticket concept using the object-oriented paradigm. As constructor argument, it expects an event collection for building its state representation. This is done by consecutively invoking the function `#applyEvent()`, which updates internal data accordingly. The operations `open()`, `comment()` and `close()` implement the defined use cases. All of them follow a similar pattern. They perform invariant protection, create events and execute the operation `#applyNewEvent()`. One exception is the function `comment()`, which also conditionally produces a “`SupportTicketReOpened`” event. The operation `#applyNewEvent()` updates the event collection and invokes the function `#applyEvent()`. Retrieving new events on the outside is done via the accessor `get newEvents()`. One notable aspect is that the component only maintains the state information it requires, without accommodating for persistence purposes.



Ticket resolutions

The Domain Model description states that support tickets can either be solved or considered invalid. This aspect is implemented with the `resolution` attribute that is expected as parameter when closing a ticket. Its possible values are “`solved`” and “`closed`”.

The next example illustrates the usage of the component ([run code](#)):

Support ticket system: Usage of object-oriented approach

```
const ticketId = generateId();
const issuerId = generateId(), supporterId = generateId();

const supportTicket = new SupportTicket();
supportTicket.open({id: ticketId, authorId: issuerId,
    title: 'My computer is broken', description: 'There is a black screen'});
supportTicket.comment({authorId: supporterId, message: 'Is it plugged in?'});
supportTicket.comment({authorId: issuerId, message: 'Oh, now it works!'});
supportTicket.close({resolution: 'solved'});

const events = supportTicket.newEvents;
const rebuiltSupportTicket = new SupportTicket(events);
rebuiltSupportTicket.comment(
    {authorId: issuerId, message: 'Hold on, it is broken again'});

console.log(rebuiltSupportTicket.newEvents);
```

The code starts with defining identifiers for a ticket, an issuer and a supporter. Then, the class `SupportTicket` is instantiated. This is followed by executing the use cases of opening a ticket, adding two comments and closing the ticket again. As previously explained, one advantage of the object-oriented approach is the continuous state update that allows consecutive actions. Next, all new events are retrieved and the class `SupportTicket` is instantiated again using them as argument. Another comment is added to the rebuilt ticket instance. Finally, all new events are retrieved and logged to the console. Executing this code shows that the last comment addition produces both a “`SupportTicketReOpened`” and a “`SupportTicketCommented`” event. Overall, the component usage is almost equal to an implementation without Event Sourcing.



Creation vs. reconstitution

One additional downside of an object-oriented approach is the complexity associated with differentiating between creation and reconstitution. The issue is that the act of performing a state mutation differs from the process of rebuilding state information. While there are multiple solution approaches for dealing with this aspect, it typically further increases the complexity of the implementation.

Separated state

As variation to the first approach, the behavioral parts and the state representation can be strictly separated from each other. [Vernon, p. 547] suggests to “split the [...] implementation into two distinct classes [...], with the state object being held by the behavioral”. When executing a use case operation, the state object is replaced with a newer version that incorporates all required updates. The advantage of this variation is that it provides a clean separation of behavior and state. In fact, it represents a stepping stone towards a more functional approach. However, this aspect can also be considered a disadvantage. While the implementation seems more functional due to immutable state objects, it faces the same issues as the previous approach.

The next code shows the relevant parts of a reworked support ticket Aggregate type ([run code usage](#)):

Support ticket system: Support Ticket state separation

```
#state = {};
#newEvents = [];

constructor(events = []) {
    events.forEach(event => this.#applyEvent(event));
}

#applyEvent({type, data}) {
    if (type === SupportTicketOpenedEvent.type)
        this.#state = {...this.#state, ticketId: data.ticketId, isOpen: true};
    if (type === SupportTicketClosedEvent.type)
        this.#state = {...this.#state, isOpen: false};
    if (type === SupportTicketReOpenedEvent.type)
        this.#state = {...this.#state, isOpen: true};
}
```

The reworked class `SupportTicket` declares the private variable `#state` and initializes it with an empty object. The function `#applyEvent()` replaces the state object in response to events instead of mutating individual private fields. This is done by creating a new object, cloning the existing state representation and updating selected properties. All the behavioral functions that require to access state information are adjusted accordingly. As explained earlier, this variant can be considered a stepping stone towards a functional approach. The immutable state objects suggest that the implementation is less error-prone due to a lower chance of unintended side effects. However, since this aspect is mainly an internal implementation detail, it has no noticeable effect on the actual component usage.

Functional approach

The Functional Programming paradigm is also a suitable option for an event-sourced Write Model implementation. Even more, [Young] argues that “an event-sourced model is not object-oriented”, but in fact functional. Instead of combining behavior and state into one unit, both aspects are kept side by side. The mechanism for creating a state representation from an event collection is defined as standalone operation. Every behavior is also implemented as free function that expects a state representation in addition to actual arguments. The resulting events that represent a state change are provided as return value. One advantage of this approach is that the Domain Model components only consist of side-effect-free operations without mutable state. One risk is that the implementation is more complex than an object-oriented variant.

The following example provides a functional approach for the support ticket Aggregate type:

Support ticket system: Functional Support Ticket

```
const applySupportTicketEvents = (state = {}, events = []) =>
  events.reduce(applyEvent, state);

const applyEvent = (state, {type, data}) => {
  if (type === SupportTicketOpenedEvent.type)
    return {...state, id: data.ticketId, isOpen: true};
  if (type === SupportTicketClosedEvent.type) return {...state, isOpen: false};
  if (type === SupportTicketReOpenedEvent.type) return {...state, isOpen: true};
  return {...state};
};

const openSupportTicket = ({id: ticketId, authorId, title, description}) => {
  if (title.length > 50) throw new Error('title length exceeded');
  if (description.length > 500) throw new Error('description length exceeded');
  return [new SupportTicketOpenedEvent({ticketId, authorId, title, description})];
};

const commentSupportTicket = (state, {authorId, message = ''}) => {
  const ticketId = state.id;
  const events = [];
  if (!state.isOpen) events.push(new SupportTicketReOpenedEvent({ticketId}));
  events.push(new SupportTicketCommentedEvent({ticketId, authorId, message}));
  return events;
};

const closeSupportTicket = (state, {resolution}) => {
```

```
if (!['solved', 'closed'].includes(resolution))
  throw new Error('invalid resolution');
return [new SupportTicketClosedEvent({ticketId: state.id, resolution})];
};
```

The operation `applySupportTicketEvents()` is responsible for building the required state representation. As arguments, it expects an initial state object and a list of events. Each event is individually passed to the operation `applyEvent()` together with a previous state object returned from the last call. Every invocation creates a new object that is a combination of the previous state representation and all event-related changes. The defined use cases are expressed as the functions `openSupportTicket()`, `commentSupportTicket()` and `closeSupportTicket()`. Similar to the objected-oriented approach, they all perform an invariant protection and create according events. The difference is that the events are directly returned to the caller. Note that the operations always return an array. This is because a use case may produce any number of events.

This next code shows an exemplary usage of the functional component ([run code](#)):

Support ticket system: Usage of functional approach

```
const userId = generateId(), supporterId = generateId();

let state = {};

const openEvents = openSupportTicket({id: ticketId, authorId: userId,
  title: 'My computer is broken', description: 'There is a black screen'});
state = applySupportTicketEvents(state, openEvents);

const firstCommentEvents = commentSupportTicket(state,
  {authorId: supporterId, message: 'Is it plugged in?'});
state = applySupportTicketEvents(state, firstCommentEvents);

const secondCommentEvents = commentSupportTicket(state,
  {authorId: userId, message: 'Oh, now it works!'});
state = applySupportTicketEvents(state, secondCommentEvents);

const closeEvents = closeSupportTicket(state, {resolution: 'solved'});
state = applySupportTicketEvents(state, closeEvents);

const reopeningCommentEvents = commentSupportTicket(state,
  {authorId: userId, message: 'Hold on, it is broken again'});

console.log(reopeningCommentEvents);
```

Similar to the usage of the object-oriented variant, the code starts with defining all required identifiers. Next, a mutable state variable is declared and initialized with an empty object. This is followed by a use case sequence of opening a ticket, adding comments, closing the ticket and adding another comment. As additional step after each behavior execution, the function `applySupportTicketEvents()` is invoked and the state variable is re-assigned. This is required to perform an execution of multiple consecutive actions. Other than with a stateful component, the functional approach shifts the responsibility of state management to the consumer. As final step, the events returned from the last comment addition are logged to the console. Again, this consists of both a “`SupportTicketReOpened`” and a “`SupportTicketCommented`” event.

As mentioned previously, there is no universal argument for one of the two illustrated approaches. Nevertheless, the code examples and the Sample Application of this book exclusively use the functional style.

Event Store

The **Event Store** is responsible for the persistence of event-sourced state. For that, it must provide the possibility to save new events and to retrieve existing ones in order of their persisting. Its usage makes Repositories obsolete. The stored items are categorized in **Event Streams**. As mentioned earlier, for software that applies DDD, the streams are separated by Aggregate identity. This means, one individual stream contains the state for one Aggregate. The Event Store must be accessible to both the write side and the read side of a context. However, it should not be exposed to the outside. For write use cases, events are loaded to rebuild state representations and new entries are saved. On the read side, events are consumed to build Read Models.



Global Event Store

Event Sourcing is typically applied to selected context implementations. Consequently, an Event Store is a local component that should be encapsulated within its respective boundaries. However, if a system is applying Event Sourcing as part of its macro-architecture, there can be a global store.

Basic functionality

At a minimum, an Event Store must provide one operation for reading a stream and another one for appending new events. Each stream has a unique identifier that is represented as a

simple string. The command to append new items must be able to persist multiple events in a transactional way. This is because a single write use case may produce more than one event. The collection of items to be saved within one transaction is typically referred to as **Commit**. In addition to existing metadata, an Event Store may add persistence-related information to individual records. Most commonly, each event is complemented with a number that represents the current version of the affected stream. This information can be used for Concurrency Control mechanisms.

As preparation for a store implementation, the first example provides a function to read a file with a fallback value ([run code usage](#)):

Filesystem: Read file with fallback helper

```
const readFileWithFallback = (filePath, fallback) =>
  readFile(filePath, 'utf-8').catch(() => fallback);
```

The next example provides a minimal filesystem-based Event Store component:

Event Store: Filesystem Event Store

```
class FilesystemEventStore {

  storageDirectory;

  constructor({storageDirectory}) {
    mkdirSync(storageDirectory, {recursive: true});
    Object.defineProperty(
      this, 'storageDirectory', {value: storageDirectory, writable: false});
  }

  async save(streamId, events) {
    const streamDirectory = `${this.storageDirectory}/${streamId}`;
    await mkdir(streamDirectory, {recursive: true});
    const currentVersion = await this.#getStreamVersion(streamDirectory);
    await Promise.all(events.map((event, index) => {
      const numberedEvent = {...event, number: currentVersion + index + 1};
      return writeFileAtomically(`${streamDirectory}/${numberedEvent.number}.json`,
        JSON.stringify(numberedEvent));
    }));
    const newVersion = currentVersion + events.length;
    await writeFile(`${streamDirectory}/_version`, `${newVersion}`);
  }
}
```

```
async load(streamId, {startVersion: start = 1} = {}) {
  const streamDirectory = `${this.storageDirectory}/${streamId}`;
  const currentVersion = await this.#getStreamVersion(streamDirectory);
  const events = await Promise.all(
    Array.from({length: currentVersion - start + 1}, (_, index) => readFile(
      `${streamDirectory}/${start + index}.json`, 'utf-8')).then(JSON.parse)));
  return {events, currentVersion};
}

#getStreamVersion(streamDirectory) {
  return readFileWithFallback(`${streamDirectory}/_version`, '0').then(Number);
}
}
```

The class `FilesystemEventStore` represents a minimal Event Store implementation. Its command `save()` is responsible for appending new events to a stream. First, the operation retrieves the stream version via the helper `#getStreamVersion()`. This function reads the file `_version` from the stream directory and uses 0 as fallback value. Next, the save operation iterates the given events. Each item is extended with a `number` property and persisted as a separate file. Finally, the file `_version` is updated with the latest value. The operation `load()` is responsible for loading events from a stream. In addition to a stream identifier, it accepts an optional start version, which has a default value of 1. All events that match the consequential version range are loaded and returned as ordered collection.



Transactional save behavior

The illustrated Event Store provides transactional behavior, even when saving multiple events. This is achieved by maintaining the stream version as a separate file. Regardless of directory contents, the component only considers files that are equal to or lower than the stored version. Therefore, the save operation can fail in between the persistence of multiple events without producing corrupted state.

The next example provides a simplified usage of the Event Store component ([run code](#)):

Event Store: Filesystem Event Store usage

```
const eventStore = new FilesystemEventStore({storageDirectory});

const browserTabId = generateId();

await eventStore.save(`browser-tab/${browserTabId}`,
  [{id: generateId(), type: 'BrowserTabOpened', data: {browserTabId}}]);
await eventStore.save(`browser-tab/${browserTabId}`,
  [{id: generateId(), type: 'BrowserTabClosed', data: {browserTabId}}]);
console.log(await eventStore.load(`browser-tab/${browserTabId}`));
```

Optimistic concurrency

Independent of further functionalities, an Event Store should support concurrent resource access. Similar to the Repository components introduced in Chapter 9, one option is to enable Optimistic Concurrency. With this approach, read operations are always allowed, while write operations require to be based on the latest status. Since most Event Store implementations already incorporate stream version numbers, it makes sense to re-use this information for Concurrency Control. Whenever new events are requested to be added, an expected version can be provided as additional argument. Only if the value matches with the actual stream version, the new records are stored. Otherwise, a concurrency conflict is reported. Ideally, this mechanism is optional, as there are valid use cases where stream versions are counterproductive.

The first example shows a simple error class for concurrency conflicts:

Event Store: Stream version mismatch error

```
class StreamVersionMismatchError extends Error {

  constructor({expectedVersion, currentVersion}) {
    super('StreamVersionMismatch');
    Object.assign(this, {expectedVersion, currentVersion});
  }

}
```

The next code provides a concurrency-safe Event Store component:

Event Store: Concurrency-safe Event Store

```
class ConcurrencySafeFilesystemEventStore {  
  
    storageDirectory; #saveQueueByStreamId = new Map();  
  
    constructor({storageDirectory}) {  
        mkdirSync(storageDirectory, {recursive: true});  
        Object.defineProperty(  
            this, 'storageDirectory', {value: storageDirectory, writable: false});  
    }  
  
    async save(streamId, events, {expectedVersion = undefined} = {}) {  
        if (!this.#saveQueueByStreamId.has(streamId))  
            this.#saveQueueByStreamId.set(streamId, new AsyncQueue());  
        const saveQueue = this.#saveQueueByStreamId.get(streamId);  
        const streamDirectory = `${this.storageDirectory}/${streamId}`;  
        await mkdir(streamDirectory, {recursive: true});  
        return saveQueue.enqueueOperation(async () => {  
            const currentVersion = await this.#getStreamVersion(streamDirectory);  
            if (expectedVersion != null && expectedVersion !== currentVersion)  
                throw new StreamVersionMismatchError({expectedVersion, currentVersion});  
            await Promise.all(events.map((event, index) => {  
                const numberedEvent = {...event, number: currentVersion + index + 1};  
                const eventFilename = `${streamDirectory}/${numberedEvent.number}.json`;  
                return writeFileAtomically(eventFilename, JSON.stringify(numberedEvent));  
            }));  
            const newVersion = currentVersion + events.length;  
            await writeFile(`${streamDirectory}/_version`, `${newVersion}`);  
        });  
    }  
  
    async load(streamId, {startVersion: start = 1} = {}) {  
        const streamDirectory = `${this.storageDirectory}/${streamId}`;  
        const currentVersion = await this.#getStreamVersion(streamDirectory);  
        const events = await Promise.all(  
            Array.from({length: currentVersion - start + 1}, (_, index) => readFile(  
                `${streamDirectory}/${start + index}.json`, 'utf-8').then(JSON.parse)));  
        return {events, currentVersion};  
    }  
  
    #getStreamVersion(streamDirectory) {  
        return readFileWithFallback(`${streamDirectory}/_version`, '0').then(Number);  
    }  
}
```

}

The class `ConcurrencySafeFilesystemEventStore` extends the previous Event Store with version-based Optimistic Concurrency. The field `#saveQueueByStreamId` is initialized with a Map object for `AsyncQueue` instances per stream. The command `save()` additionally accepts an optional expected version. As first step, the operation retrieves the async queue for the respective stream. Then, it defines and enqueues the actual saving process, which consists of multiple parts. First, the stream version is loaded. If an expected version is provided and differs from the stored value, a `StreamVersionMismatchError` instance is thrown. Afterwards, each event is extended with a number and persisted. Finally, the file `_version` is updated accordingly. The operation `load()` remains unchanged. Similar to the concurrency-safe Repository from Chapter 9, the component only supports concurrency when used as single instance.



Queuing save operations

Even when not providing an expected version argument, the concurrency-safe Event Store must queue all save requests per stream. This is because the saving process affects multiple transactions, one per each modified file. Executing two or more requests concurrently would risk event overwrites due to using conflicting version numbers.

The next example illustrates the usage of the concurrency-safe Event Store ([run code](#)):

Event Store: Concurrency-safe Event Store usage

```
const eventStore = new ConcurrencySafeFilesystemEventStore({storageDirectory});

const meetingId = generateId(), participantId = generateId();

await eventStore.save(`meeting/${meetingId}`,
  [{id: generateId(), type: 'MeetingScheduled', data: {meetingId}}]);
const {currentVersion} = await eventStore.load(`meeting/${meetingId}`);

const meetingParticipantAddedEvent = {id: generateId(),
  type: 'MeetingParticipantAdded', data: {meetingId, participantId}};
await eventStore.save(`meeting/${meetingId}`,
  [meetingParticipantAddedEvent], {expectedVersion: currentVersion});
await eventStore.save(`meeting/${meetingId}`,
  [meetingParticipantAddedEvent], {expectedVersion: currentVersion});
```

Retries and conflict resolution

As explained in Chapter 9 in the context of Repositories, there are mainly two action possibilities when encountering concurrency conflicts. One option is to retry the failing operation for a certain number of attempts before aborting with an error. This can be a valid approach for certain scenarios, but can also hint to an issue in the Aggregate design. Another option is to implement a custom conflict resolution mechanism. In case of event-sourced state, the functionality must compare the conflicting sets of events on top of the expected version. If their combination does not produce a logical conflict, the new events can be appended. Otherwise, the resolution mechanism can attempt to re-execute the original behavior. In case of repeated failure, it should abort with an error.

Stream subscriptions

On top of saving and loading events, an Event Store should also support stream subscriptions. The idea is to treat streams as a continuous source of events without differentiating between existing records and future items. This mechanism is especially useful for the creation of Read Models. In theory, the desired behavior can be emulated with periodical loading. However, it makes more sense for the Event Store to support it as native functionality. This keeps the responsibility within the component and avoids manual polling, which can be wasteful in terms of resources. The subscription mechanism can also support asynchronous subscribers by queuing and awaiting callback executions. However, this functionality must not be misused for outbound event distribution. Subscriptions should only be used inside the owning context.

The first example provides a component that implements an event stream subscription:

Event Store: Filesystem event stream subscription

```
class FilesystemEventStreamSubscription {  
  
    #streamDirectory; #callback; #processedVersion; #versionFilePath;  
    #callbackExecutionQueue = new AsyncQueue(); #watcher;  
  
    constructor({streamDirectory, callback, startVersion}) {  
        this.#streamDirectory = streamDirectory;  
        this.#callback = callback;  
        this.#processedVersion = startVersion - 1;  
        this.#versionFilePath = `${this.#streamDirectory}/_version`;  
    }  
}
```

```
async initialize() {
    await mkdir(this.#streamDirectory, {recursive: true});
    await writeFile(this.#versionFilePath, '', {flag: 'wx'}).catch(() => {});
    this.#watcher = watch(this.#versionFilePath, () => this.#processNewEvents());
    this.#processNewEvents();
}

#processNewEvents() {
    this.#callbackExecutionQueue.enqueueOperation(async () => {
        const availableVersion = Number(
            await readFileSync(this.#versionFilePath, '0'));
        while (this.#processedVersion < availableVersion) {
            const event = JSON.parse(await readFile(
                `${this.#streamDirectory}/${++this.#processedVersion}.json`, 'utf-8'));
            await this.#callback(event);
        }
    });
}

destroy() { this.#watcher.close(); }

}
```

The class `FilesystemEventStreamSubscription` represents a subscription mechanism for event streams. Its constructor expects a stream directory, a callback and a start version. Also, it defines a private variable for the currently processed version and one `AsyncQueue` instance. The operation `initialize()` ensures the existence of the stream version file, creates a file watcher and invokes the function `#processNewEvents()`. This operation is also executed by the watcher callback. Its responsibility is to enqueue notifications for all available unprocessed events. For that purpose, it first retrieves the latest stream version. Then, for each value within the version range of unprocessed items, it loads the according event and invokes the subscription callback. The iteration supports asynchronous subscribers by awaiting the callback completion. The command `destroy()` closes the file watcher.



Watching non-existent files

The filesystem watcher API `fs.watch()` does not work with files that are only created later. Therefore, the illustrated Event Store component creates the version file if it does not exist. This ensures that the `fs.watch()` call always operates on an existing file.

The next example provides an Event-Store with a stream subscription mechanism:

Event Store: Event Store with subscriptions

```
class FilesystemEventStoreWithSubscription
  extends ConcurrencySafeFilesystemEventStore {

  #subscriptionByCallback = new Map();

  async subscribe(streamId, callback, {startVersion = 1} = {}) {
    const streamDirectory = `${this.storageDirectory}/${streamId}`;
    const subscription = new FilesystemEventStreamSubscription(
      {streamDirectory, callback, startVersion});
    await subscription.initialize();
    this.#subscriptionByCallback.set(callback, subscription);
  }

  unsubscribe(callback) {
    this.#subscriptionByCallback.get(callback).destroy();
    this.#subscriptionByCallback.delete(callback);
  }
}
```

The class `FilesystemEventStoreWithSubscription` extends the concurrency-safe filesystem-based Event Store component with a stream subscription mechanism. For maintaining subscriptions and callbacks, it defines the private field `#subscriptionByCallback` and initializes it with a `Map` instance. The operation `subscribe()` expects a stream identifier, a callback operation and an optional start version, which defaults to 1. With the given arguments, it first creates a `FilesystemEventStreamSubscription` instance. Then, the function `initialize()` of the new subscription is executed. Lastly, an according entry is added to the map object. The command `unsubscribe()` is responsible for deleting a subscription. Upon execution, the target object is retrieved using the provided callback as key and its `destroy()` function is invoked.

The usage of the Event Store with stream subscriptions can be seen in the following example ([run code](#)):

Event Store: Event Store with subscriptions usage

```
const eventStore = new FilesystemEventStoreWithSubscription({storageDirectory});

const guestbookId = generateId();

await eventStore.save(`guestbook/${guestbookId}`,
  [{id: generateId(), type: 'GuestbookMessageWritten',
    data: {guestbookId, entryId: generateId(), message: 'Hello!'}}]);
await eventStore.subscribe(`guestbook/${guestbookId}`, console.log);
await timeout(1000);
await eventStore.save(`guestbook/${guestbookId}`,
  [{id: generateId(), type: 'GuestbookMessageWritten',
    data: {guestbookId, entryId: generateId(), message: 'Great page'}}]);
```

Global event stream

For processing selected event types across individual stream boundaries, an Event Store component can provide categorized or global streams. In terms of DDD, this allows to subscribe for events across multiple Aggregates. One simple possibility is to create a single global stream and continuously append all records of every existing stream. This variant has a low complexity for the Event Store but can cause a wasteful consumption. Another option is to maintain one grouped stream per event type. Naturally, the implementation results in more complexity within the Event Store, but allows for more targeted processing. Both grouped and global streams should guarantee an internal fixed event order. At the same time, an Event Store should not try to establish a global order across different streams.

The next example implements a class for an Event Store component with a global stream:

Event Store: Event Store with global stream

```
class FilesystemEventStoreWithGlobalStream
  extends FilesystemEventStoreWithSubscription {

  #subscribedStreams = [];

  async save(streamId, events) {
    if (streamId !== '$global' && !this.#subscribedStreams.includes(streamId))
      await this.#linkStreamToGlobalStream(streamId);
    return super.save(streamId, events);
  }
}
```

```
async #linkStreamToGlobalStream(streamId) {
    this.#subscribedStreams.push(streamId);
    const streamVersionDirectory =
        `${this.storageDirectory}/$global/stream-versions/${streamId}`;
    await mkdir(streamVersionDirectory, {recursive: true});
    const processedVersion = Number(
        await readFileSync(`${streamVersionDirectory}/_version`, '0'));
    await this.subscribe(streamId, async event => {
        const metadata = {...event.metadata,
            originStreamId: streamId, originNumber: event.number};
        const events = [...event, number: undefined, metadata];
        await this.save('$global', events);
        await writeFile(`${streamVersionDirectory}/_version`, `${event.number}`);
    }, {startVersion: processedVersion + 1});
}

}
```

The class `FilesystemEventStoreWithGlobalStream` extends the previously illustrated component with a global stream. Upon construction, it defines an array for subscribed stream identifiers. When executing the operation `save()`, the component checks whether itself is subscribed to the target stream. If not, the function `#linkStreamToGlobalStream()` is executed. Afterwards, the original `save()` operation of the parent class is invoked. The command `#linkStreamToGlobalStream()` creates a subscription that appends incoming events to the grouped stream “`$global`”. For tracking the processed version of a stream, an individual `_version` file is maintained. Every incoming event is cloned, its number is removed and the metadata is extended with the properties `originStreamId` and `originNumber`. Afterwards, the event is appended to the stream “`$global`” without expected version. Finally, the new processed stream version is persisted.



Global stream integrity

The global stream of the illustrated Event Store component provides an “at least once” guarantee. This means that every event from each stream eventually appears. However, it can happen that an individual event appears more than once. This is because the persisting of an event together with the update of the processed stream version is not transactional.

The component is complemented with an exemplary usage code ([run code](#)):

Event Store: Event Store with global stream usage

```
const eventStore = new FilesystemEventStoreWithGlobalStream({storageDirectory});  
  
const userId = generateId();  
  
eventStore.subscribe('$global', console.log);  
await eventStore.save(`user/${userId}`,  
  [{id: generateId(), type: 'UserLoggedIn', data: {userId}}]);  
await eventStore.save(`user/${userId}`,  
  [{id: generateId(), type: 'UserLoggedOut', data: {userId}}]);
```

Read Model Projection

Read Model Projections are components that consume event streams and compute derived read data. Their functionality is similar to a Read Model synchronization with Domain Events as described in Chapter 11. The key difference is that projections consume events from an Event Store. In general, the facilitation of this process is a combination of infrastructural aspects and Application Layer components. However, the specialized transformation of events into derived data architecturally belongs to the Domain part. For simple use cases, it is acceptable to keep the respective code directly in the Application Layer. In contrast, complex transformations should be implemented as dedicated Domain components. Projections can either be on-demand computations or continuous processes. Both of the types are described and illustrated in the following subsections.



Projection versus Projection

The term “projection” can be ambiguous. For one, it can stand for the result of a process. Secondly, it can refer to the process of creating a result. This book uses the term “projection” for the act of projecting, not for the result itself. As this process can incorporate multiple parts, the term does not strictly refer to one specific component.

On-demand projections

Projections can be on-demand one-time operations that retrieve events, perform transformations and respond with derived data. Upon execution, the events from all affected streams are

loaded, the domain-specific transformations are applied and the result is returned. Typically, the computed Read Model data is considered transient and therefore not retained. This approach is useful for specialized read use cases that require up-to-date information and are executed infrequently. For a generic and re-usable projection mechanism, selected parts of the transformation logic can even be parametrized. One potential downside of this approach is its performance implications. All required events must be loaded from the Event Store and the transformation logic must be executed for all of them. Most commonly, on-demand projections are used for static reporting data.

Example: Playtime statistics

Consider implementing a projection component for the calculation of playtime statistics as part of a gaming platform. The required behavior is to determine the overall time an individual person spent in games. Consequently, the player identity should be a parameter to the projection. The derived data can be used for personal reporting as well as to understand the popularity of games. The event-sourced Write Model for a player incorporates many possible state changes. For the example, the two relevant types are “PlayerStartedPlayingGame” and “PlayerStoppedPlaying”. Both of them enclose a player identifier and the time of occurrence as custom data. In addition, the start of a game also includes its identity. Note that the time is not considered metadata, as it represents essential domain-specific information.

The first example provides the relevant event definitions for the player component:

Playtime statistics: Event definitions

```
const PlayerStartedPlayingGameEvent = createEventType('PlayerStartedPlayingGame',
  {playerId: 'string', gameId: 'string', time: 'number'});

const PlayerStoppedPlayingEvent = createEventType('PlayerStoppedPlaying',
  {playerId: 'string', time: 'number'});
```

The next code implements the projection component to determine playtime statistics:

Playtime statistics: Projection

```
class PlaytimeStatisticsProjection {  
  
    #eventStore;  
  
    constructor({eventStore}) {  
        this.#eventStore = eventStore;  
    }  
  
    async getPlaytime(playerId) {  
        let result = 0, lastStart = 0;  
        const {events} = await this.#eventStore.load(`player/${playerId}`);  
        events.forEach(({data, type}) => {  
            if (type === PlayerStartedPlayingGameEvent.type) lastStart = data.time;  
            if (type === PlayerStoppedPlayingEvent.type) result += data.time - lastStart;  
        });  
        return result;  
    }  
}
```

The types `PlayerStartedPlayingGameEvent` and `PlayerStoppedPlayingEvent` define the relevant state transitions for a player. The class `PlaytimeStatisticsProjection` represents an on-demand projection component. Its constructor expects an Event Store instance. The function `getPlaytime()` computes the total playtime for a given player identity. As first step, the operation defines mutable variables for the overall result and the last respective start of a game. Next, it loads the according event stream and iterates the returned collection. For each “`PlayerStartedPlayingGame`” event, the variable `lastStart` is set to the event time. Every “`PlayerStoppedPlaying`” event causes to add the difference between the saved start and the stop time to the overall result. Finally, the accumulated value is returned. For this example, the domain-specific transformation is directly placed in the Application Layer component.

The last example shows an exemplary usage code of the previous components ([run code](#)):

Playtime statistics: Usage

```
const eventStore = new FilesystemEventStore({storageDirectory});

const playerId = generateId(), game1Id = generateId(), game2Id = generateId();

const events = [
  new PlayerStartedPlayingGameEvent({playerId, gameId: game1Id, time: 0}),
  new PlayerStoppedPlayingEvent({playerId, time: 1000}),
  new PlayerStartedPlayingGameEvent({playerId, gameId: game2Id, time: 4000}),
  new PlayerStoppedPlayingEvent({playerId, time: 5000}),
  new PlayerStartedPlayingGameEvent({playerId, gameId: game1Id, time: 6000}),
  new PlayerStoppedPlayingEvent({playerId, time: 10000}),
];

await eventStore.save(`player/${playerId}`, events);
const projection = new PlaytimeStatistics({eventStore});
console.log(await projection.getPlaytime(playerId));
```

The code starts with creating an instance of the class `FilesystemEventStore`. Next, it defines variables for a player identity and for two game identifiers. Then, an array of events is created that contains three exemplary pairs of “PlayerStartedPlayingGame” and “PlayerStoppedPlaying” events. As next step, the event collection is appended to the affected player stream in the Event Store. Afterwards, the class `PlaytimeStatistics` is instantiated. Finally, the operation `getPlaytime()` is invoked with the player identity as argument and its result is logged to the console. Executing the code outputs the accumulated playtime of all three games for the affected player. Note that the performance of this computation decreases with the number of events. However, this is negligible as the playtime is likely to be requested infrequently.

Continuous projections

Projections can also be long-running operations that consume event notifications and continuously update Read Model data. Upon initialization, they create subscriptions for all associated streams of an Event Store. The registered callback operations are responsible for invoking the transformation logic that adjusts the derived data structures. Effectively, this mechanism facilitates eventually consistent Read Models that are continuously updated in response to state changes. When a read use case is executed, the data is directly retrieved from its storage without interacting with the projection. This approach is useful for frequently accessed information and for large event streams, where on-demand computations can be

too expensive. Typically, continuous projections subscribe to categorized or global streams and maintain Read Models that are part of a running software.

Example: Warehouse stock

Consider implementing a projection component for stock Read Models of warehouse goods. The required functionality is to query the current stock of an individual good inside a warehouse. This information can be displayed in other software such as a shopping platform to indicate the availability of a product. However, as the returned data is eventually consistent, it must only be used for nonbinding informational purposes. For the example, the three relevant state transitions are represented by the event types “GoodRegistered”, “GoodStocked” and “GoodDestocked”. The first one describes the initial admission of a certain good and contains an identifier and a title as data. The other types indicate that the stock of a good has changed and enclose its identity and a units value.

The first example provides the event definitions for the good component:

Warehouse stock: Event definitions

```
const GoodRegisteredEvent = createEventType('GoodRegistered',
  {goodId: 'string', title: 'string'});

const GoodStockedEvent = createEventType('GoodStocked',
  {goodId: 'string', units: 'number'});

const GoodDestockedEvent = createEventType('GoodDestocked',
  {goodId: 'string', units: 'number'})
```

The second example implements a domain-specific operation to update a stock Read Model:

Warehouse stock: Stock update

```
const updateStock = (state, {type, data}) => {
  if (type === GoodRegisteredEvent.type)
    return {...state, goodId: data.goodId, title: data.title, stock: 0};
  if (type === GoodStockedEvent.type)
    return {...state, stock: state.stock + data.units};
  if (type === GoodDestockedEvent.type)
    return {...state, stock: state.stock - data.units};
  return {...state};
};
```

This is followed by a projection component to maintain stock Read Models:

Warehouse stock: Continuous Projection

```
class StockProjection {  
  
    #eventStore; #stockStorage;  
  
    constructor({eventStore, stockStorage}) {  
        this.#eventStore = eventStore;  
        this.#stockStorage = stockStorage;  
    }  
  
    activate() {  
        this.#eventStore.subscribe('$global', event =>  
            this.#stockStorage.set(event.data.goodId,  
                updateStock(this.#stockStorage.get(event.data.goodId) || {}, event)));  
    }  
}
```

The three types `GoodRegisteredEvent`, `GoodStockedEvent` and `GoodDestockedEvent` define the possible state changes for a warehouse good. In contrast to the example of the previous subsection, the domain-specific transformation is implemented as the dedicated operation `updateStock()`. This computation expects a data structure together with an event and returns an updated version of the data. In fact, the function is similar to the `applyEvent()` operation of functional Write Model components. The projection class `StockProjection` is responsible for maintaining stock Read Models. Its constructor expects an Event Store and a storage component. The operation `activate()` sets up the projection by subscribing to the global stream of the Event Store. Upon notification, the callback retrieves the affected stock Read Model, passes it to the function `updateStock()` and saves the result.

The last example illustrates the usage of all components ([run code](#)):

Warehouse stock: Usage

```
const eventStore = new FilesystemEventStoreWithGlobalStream(
  {storageDirectory: `${storageDirectory}/event-store`});
const stockStorage = new Map();
const projection = new StockProjection({eventStore, stockStorage});

const goodId = 'good-id';
const streamId = `good/${goodId}`;

await projection.activate();
const {currentVersion} = await eventStore.load(streamId);
if (currentVersion === 0) await eventStore.save(
  streamId, [new GoodRegisteredEvent({goodId, title: 'USB Charger (5v / 2A)'})]);
await eventStore.save(streamId, [new GoodStockedEvent({goodId, units: 100}),
  new GoodDestockedEvent({goodId, units: 95})]);
await timeout(125);
console.log(stockStorage.get(goodId));
```

As first step, the code instantiates the component `FilesystemEventStoreWithGlobalStream`. Afterwards, it defines a `Map` object as in-memory storage. Then, an instance of the class `StockProjection` is created, using the previous components as arguments. This is followed by the definition of variables for a static good identity and a stream identifier. Next, the projection is initialized and the current global stream version is retrieved. If its value is 0, a new “GoodRegistered” event is appended to the respective stream. Furthermore, a pair of “GoodStocked” and “GoodDestocked” events is created and saved. Lastly, the stock Read Model is retrieved and logged to the console. Executing the code outputs a stock Read Model with five units. With every further execution, the events accumulate and the data updates accordingly.

Transient vs persistent data

Projections can maintain either transient or persistent data. Read Models that are stored in-memory can only be accessed from their enclosing runtime and must be rebuilt upon restarting. Persistent data enables inter-process access and incremental updates independent of program lifespans. However, it requires to persistently track the processed version of each affected stream. In addition to saving updated data, the last processed event number must be stored. This enables a stream subscription to always restart from its last processed version. Ideally, the information can be saved together with the Read Model as one transaction. Otherwise, the projection facilitates an “at least once” processing guarantee and may receive

duplicates. Typically, this necessitates idempotent transformations, where processing one event repeatedly has the same effect as done once.

Example: Warehouse stock extension

Consider extending the projection mechanism for stock Read Models of warehouse goods. While the previous solution works correctly, the approach of storing data exclusively in-memory can cause performance issues. In general, a warehouse software can be expected to be a long-running system without in-between restarts or crashes. However, as with any other software, extensions and bug-fixes must be performed at times and typically demand a restart. For warehouses with millions of goods, each with frequent stock changes, recreating Read Models on every restart is not feasible. One solution approach is to persistently store the derived stock data and track the processed version of the global stream. This makes it possible for the projection to skip already processed events after a required restart.

The following example shows a persistent projection component for stock Read Models:

Warehouse stock: Persistent projection

```
class PersistentGoodStockProjection {

    #eventStore; #stockStorage; #versionFilePath;

    constructor({eventStore, stockStorage, versionDirectory}) {
        this.#eventStore = eventStore;
        this.#stockStorage = stockStorage;
        this.#versionFilePath = `${versionDirectory}/_processed-stream-version`;
        mkdirSync(versionDirectory, {recursive: true});
    }

    async activate() {
        const processedVersion =
            Number(await readFileWithFallback(this.#versionFilePath, '0'));
        this.#eventStore.subscribe('$global', async event => {
            const {number, data} = event;
            if (!data.goodId) return;
            const stock = await this.#stockStorage.load(data.goodId).catch(() => ({}));
            await this.#stockStorage.save(data.goodId, updateStock(stock, event));
            await writeFile(this.#versionFilePath, `${number}`);
        }, {startVersion: processedVersion + 1});
    }
}
```

The class `PersistentStockProjection` is responsible for maintaining persistent stock Read Models. Its constructor expects an Event Store, a storage component and a version directory. Similar to the in-memory counterpart, the function `activate()` sets up the projection. First, it loads the last processed stream version with a fallback value of 0. Then, it subscribes to the global event stream, using the processed version incremented by 1 as `startVersion` parameter. The registered callback first checks whether a received event contains a good identifier and returns if none is found. Otherwise, the according stock Read Model is loaded with an empty object as fallback. Afterwards, the data and the event are passed to the function `updateStock()` and its result is persisted. Finally, the processed stream version is updated.



Chances of event duplicates

The illustrated projection tracks the processed stream version in a separate file apart from the Read Model. This means that the combination of persisting a data structure and storing the latest processed version is not transactional. Consequently, the projection may receive duplicates after an interruption between the two steps. Again, this circumstance represents an “at least once” processing guarantee.

The last example shows an exemplary usage code of the previous components ([run code](#)):

Warehouse stock: Usage with persistent projection

```
const eventStore = new FilesystemEventStoreWithGlobalStream(
  {storageDirectory: `${storageDirectory}/event-store`});
const stockStorage = new JSONFileStorage(`${storageDirectory}/stock`);
const projection = new PersistentGoodStockProjection(
  {eventStore, stockStorage, versionDirectory: `${storageDirectory}/projection`});

const goodId = 'another-good-id';
const title = 'USB Charger (5v / 2A)';
const streamId = `good/${goodId}`;

await projection.activate();
const {currentVersion} = await eventStore.load(streamId);
if (currentVersion === 0) await eventStore.save(
  streamId, [new GoodRegisteredEvent({goodId, title})]);
await eventStore.save(streamId, [new GoodStockedEvent({goodId, units: 100}),
  new GoodDestockedEvent({goodId, units: 95})]);
await timeout(125);
console.log(await stockStorage.load(goodId));
```

The code is very similar to the in-memory projection approach. Two notable differences are the usages of the component `JSONFileStorage` and the class `PersistentGoodStockProjection` as projection. Other than that, the overall flow is almost identical. Also, executing the code produces the same output as with the previous approach. Even more, additional executions have the same effect of accumulating the stock value. Overall, both approaches have their advantages and disadvantages. The in-memory variant can have a better performance when there are not many stored events. This is because there is no computational overhead due to persistence. However, as soon as a warehouse reaches a certain size, it makes more sense to use persistent projections. They help to ensure faster startup times.

Domain Event publishing

Although Event-Driven Architecture and Event Sourcing share similarities, the two concepts should be clearly differentiated. EDA is a style that is commonly applied on a system level. Its purpose is to enable loosely-coupled messaging via an Event Bus. In contrast, Event Sourcing is best applied to individual software areas. The events are persisted in an Event Store and sent as notifications to local subscribers. This mechanism should not be misused for system-wide event distribution. When applying both architectural concepts, a dedicated subscriber should actively forward selected or derived information to a separate Event Bus. Ideally, this is combined with a mechanism that is responsible for converting Event Sourcing records to Domain Events. In terms of CQRS, the process should be part of the write side.

The following code provides a Domain Event publisher component that integrates Event Sourcing events with Domain Event publishing:

Event Store: Domain Event Publisher

```
class DomainEventPublisher {  
  
    #eventStore; #eventBus; #publicEventTypes; #versionFilePath;  
  
    constructor({storageDirectory, eventStore, eventBus, publicEventTypes}) {  
        this.#eventStore = eventStore;  
        this.#eventBus = eventBus;  
        this.#publicEventTypes = publicEventTypes;  
        this.#versionFilePath = `${storageDirectory}/_processed-stream-version`;  
        mkdirSync(storageDirectory, {recursive: true});  
    }  
  
    async activate() {  
        const processedVersion =
```

```
Number(await readFileWithFallback(this.#versionFilePath, '0'));
this.#eventStore.subscribe('$global', async event => {
  const shouldPublishEvent = this.#publicEventTypes.includes(event.type);
  if (shouldPublishEvent) await this.#eventBus.publish(event);
  await writeFile(this.#versionFilePath, `${event.number}`);
}, {startVersion: processedVersion + 1});
}

}
```

The class `DomainEventPublisher` is a component that selectively forwards Event Sourcing records to an Event Bus. Its name is identical to a related functionality introduced in Chapter 9. As arguments, it expects a storage directory, an Event Store, an Event Bus and an array of public event types. Its function `activate()` configures the forwarding mechanism. First, it retrieves the last processed stream version with 0 as fallback value. Then, it subscribes to the global event stream. For each incoming event, the subscription callback tests whether its type is considered public. If that is the case, the event is published via the Event Bus. Finally, the processed stream version is persisted. Similar to previously illustrated persistent projections, the component establishes an “at least once” delivery guarantee.

The following example shows an exemplary usage of the Domain Event publisher ([run code](#)):

Domain Event Publisher: Usage

```
const eventStore = new EventStore(
  {storageDirectory: `${rootStorageDirectory}/event-store`});

const eventBus = new EventBus();

const publicEventTypes = ['UserCreated', 'UserRemoved'];

const domainEventPublisher = new DomainEventPublisher(
  {eventStore, eventBus, publicEventTypes,
   storageDirectory: `${rootStorageDirectory}/domain-event-publisher`});
domainEventPublisher.activate();

const userId = generateId();

eventBus.subscribe('UserCreated', console.log);
eventBus.subscribe('UserRemoved', console.log);

await eventStore.save(`user/${userId}`,
  [{id: generateId(), type: 'UserCreated', data: {userId}}]);
```

```
await eventStore.save(`user/${userId}`,  
  [{id: generateId(), type: 'UserLoggedIn', data: {userId}}]);  
await eventStore.save(`user/${userId}`,  
  [{id: generateId(), type: 'UserRemoved', data: {userId}}]);
```

Sample Application: Event Sourcing

This section illustrates the introduction of Event Sourcing to the Sample Application implementation. Multiple architectural parts of the existing source code require to be adapted. For the persistence of state, every context maintains a separate Event Store. This makes all custom Repositories and factories obsolete. The existing Domain Event definitions are transformed into full sets of state transitions. Each Write Model component is refactored to an event-sourced implementation using a functional approach. The Command Handlers are adjusted accordingly. On the read side, each Read Model synchronization is replaced with a continuous projection component. Similar to the previous chapter, this section only shows the most distinct parts of the modified source code. Even more, Commands, Queries and Query Handlers are excluded completely, as they remain unmodified.

Shared code and patterns

For the implementation of Event Sourcing, the Sample Application requires further generic functionalities. The previously illustrated Event Store implementations are combined into a single infrastructural component. The resulting class `ConcurrencySafeFilesystemEventStore` is an Event Store that is concurrency-safe, provides stream subscriptions and exposes a global event stream. Due to its Concurrency Control mechanism, Command Handlers can retain their support for concurrent access. The store component requires to also introduce the function `readFileWithFallback()` as well as the error type `StreamVersionMismatchError`. For selectively publishing events via an Event Bus, the class `DomainEventPublisher` is re-used. Note that there is no re-usable code required for event-sourced Write Model components or Read Model projections. The following implementations apply the relevant patterns from the previous sections of this chapter.

User context

The user context implementation requires the least complex modifications to be made for correctly apply Event Sourcing. Overall, this part contains four Domain components, of

which only the user represents an Aggregate type. The other components are Value Objects. While the user Aggregate contains numerous behavioral operations, most of them are similar and exclusively update an individual attribute. The event types of the context represent an almost complete set of possible state transitions. In addition to that, none of them must be forwarded as Domain Events to an Event Bus. One distinct aspect of the context is the e-mail registry component and its according synchronization mechanism. However, refactoring this part to work with Event Sourcing is straight forward and requires only minimal changes.

The first example provides the reworked event definitions:

User context: Events

```
const UserCreatedEvent = createEventType('UserCreated', {userId: 'string',
  username: 'string', password: 'string', emailAddress: 'string', role: 'string'});
const UsernameChangedEvent = createEventType(
  'UsernameChanged', {userId: 'string', username: 'string'});
const UserPasswordChangedEvent = createEventType(
  'UserPasswordChanged', {userId: 'string', password: 'string'});
const UserEmailAddressChangedEvent = createEventType(
  'UserEmailAddressChanged', {userId: 'string', emailAddress: 'string'});
const UserRoleChangedEvent = createEventType(
  'UserRoleChanged', {userId: 'string', role: 'string'});
```

The next code shows the event-sourced implementation of the user Entity type:

User context: User Write Model

```
const applyUserEvents = (state, events) => events.reduce(applyEvent, state);

const applyEvent = (state, event) => {
  if (event.type === UserCreatedEvent.type)
    return {...state, id: event.data.userId, password: event.data.password};
  if (event.type === UserPasswordChangedEvent.type)
    return {...state, password: event.data.password};
  return {...state};
};

const createUser = data => {
  const {id, username, emailAddress, password, role, emailAvailability} = data;
  verify('valid id', id != null);
  verify('valid username', typeof username == 'string' && !username);
  verify('valid e-mail address', emailAddress instanceof EmailAddress);
  verify('unused e-mail', emailAvailability.isEmailAvailable(emailAddress));
  verify('valid password', typeof password == 'string' && !password);
```

```
verify('valid role', role.constructor === Role);
return [new UserCreatedEvent({userId: id, username,
    emailAddress: emailAddress.value, password, role: role.name})];
};

const updateUsername = (state, username) => {
    verify('valid username', typeof username === 'string' && !!username);
    return [new UsernameChangedEvent({userId: state.id, username})];
};

const updateEmailAddress = (state, emailAddress, emailAvailability) => {
    verify('valid e-mail address', emailAddress instanceof EmailAddress);
    verify('unused e-mail', emailAvailability.isEmailAvailable(emailAddress));
    return [new UserEmailAddressChangedEvent(
        {userId: state.id, emailAddress: emailAddress.value})];
};

const updatePassword = (state, password) => {
    verify('valid password', typeof password === 'string' && !!password);
    return [new UserPasswordChangedEvent({userId: state.id, password})];
};
```

The event definitions are extended with two aspects. For one, the type `UserCreatedEvent` expects an additional `password` field. Secondly, the type `UserPasswordChangedEvent` is introduced. For the Write Model part, the previously existing class `User` is replaced with a functional event-sourced variant. The operation `applyUserEvents()` is responsible for building a state representation and internally executes the function `applyEvent()` repeatedly. Creating a new user is done with the command `createUser()`. Modifying an attribute of an existing Aggregate is achieved with the respective update command. In fact, the event-sourced user component is shorter compared to the implementation without Event Sourcing. One reason is that there is no stateful structure involved. Also, the Domain Model component does not require any accommodations for persistence such as getter functions.

The following example shows the most relevant parts of the refactored Command Handlers class:

User context: Command Handlers

```
class UserCommandHandlers {  
  
    #eventStore; #hashPassword; #emailRegistry;  
  
    constructor({eventStore, emailRegistry, hashPassword}) {  
        this.#eventStore = eventStore;  
        this.#hashPassword = hashPassword;  
        this.#emailRegistry = emailRegistry;  
    }  
  
    handleCommand = createMessageForwarder(this, {messageSuffix: 'Command'});  
  
    async handleCreateUserCommand(command) {  
        const {data: {userId, username, emailAddress, password, role}} = command;  
        const events = createUser({id: userId, username,  
            password: this.#hashPassword(password), role: new Role(role),  
            emailAddress: new EmailAddress(emailAddress),  
            emailAvailability: this.#emailRegistry});  
        await this.#eventStore.save(`user/${userId}`, events, {expectedVersion: 0});  
    }  
  
    async handleLoginUserCommand({data: {userId, password}}) {  
        const {events} = await this.#eventStore.load(`user/${userId}`);  
        const state = applyUserEvents({}, events);  
        if (state.password !== this.#hashPassword(password))  
            throw new Error('invalid password');  
    }  
  
    async handleUpdateUserEmailAddressCommand({data: {userId, emailAddress}}) {  
        const {events, currentVersion} = await this.#eventStore.load(`user/${userId}`);  
        const newEvents = updateEmailAddress(applyUserEvents({}, events),  
            new EmailAddress(emailAddress), this.#emailRegistry);  
        await this.#eventStore.save(  
            `user/${userId}`, newEvents, {expectedVersion: currentVersion});  
    }  
  
    /* .. handleUpdateUsernameCommand() .. */  
    /* .. handleUpdateUserPasswordCommand() .. */  
    /* .. handleUpdateUserRoleCommand() .. */  
}
```

The class `UserCommandHandlers` is adapted to work with an Event Store and an event-sourced Write Model. Each handler operation follows the same pattern. First, the affected event stream is loaded. Naturally, the operation `handleCreateUserCommand()` skips this step. Next, a state representation is built by invoking the function `applyUserEvents()` with the retrieved events. Then, the target behavior is executed with the state and the command arguments as input. Finally, the new events are appended to the affected stream. As part of every change operation, the current stream version is retrieved and later used as `expectedVersion` parameter upon saving. This is done to enforce Optimistic Concurrency. If multiple change operations to the same Aggregate are issued concurrently, one fails due to a version mismatch.

Apart from the Command Handlers, other selected components in the Application Layer of the user context are also adjusted. The class `UserDomainEventHandlers` is refactored to the projection `EmailRegistryProjection` for maintaining the e-mail registry. Instead of subscribing to an Event Bus, it consumes the global event stream. On the read side, the class `UserReadModelSynchronization` is refactored to a projection and named `UserReadModelProjection`. Finally, the Query Handlers component remains unmodified.

The last example of this subsection provides an example usage of the user context ([run code](#)):

User context: Overall usage

```
const eventStore = new ConcurrencySafeFilesystemEventStore(
  {storageDirectory: `${rootStorageDirectory}/event-store`});

const emailRegistry = new EmailRegistry();

const userCommandHandlers = new UserCommandHandlers(
  {eventStore, emailRegistry, hashPassword: createMd5Hash});
const emailRegistryProjection = new EmailRegistryProjection(
  {emailRegistry, eventStore});
emailRegistryProjection.activate();

const userReadModelStorage = new InMemoryIndexedStorage(
  {indexes: ['emailAddress']});
const userReadModelProjection = new UserReadModelProjection(
  {userReadModelStorage, eventStore});
userReadModelProjection.activate();
const userQueryHandlers = new UserQueryHandlers({userReadModelStorage});

const randomId = generateId();
const emailAddress1 = `johnathan${randomId}@example.com`;
const emailAddress2 = `john${randomId}@example.com`;
```

```
const userId1 = generateId(), userId2 = generateId();

const commandsToExecute = [
    new CreateUserCommand({data: {userId: userId1, username: 'johnathan',
        emailAddress: emailAddress1, password: 'pw1', role: 'user'}}),
    new UpdateUserEmailAddressCommand(
        {data: {userId: userId1, emailAddress: emailAddress2}}),
];
for (const command of commandsToExecute)
    await userCommandHandlers.handleCommand(command);
await timeout(125);
const result = await userQueryHandlers.handleQuery(
    new FindUserByEmailAddressQuery({data: {emailAddress: emailAddress2}}));
console.log(result);
await userCommandHandlers.handleCommand(new CreateUserCommand({data:
    {userId: userId2, username: 'john',
        emailAddress: emailAddress2, password: 'pw1', role: 'user'}}));
```

Project context

The project context implementation requires more adaptations in order to work with Event Sourcing. Compared to the other contexts, its Domain Layer contains the highest amount of individual parts. There are four separate components, of which three represent Aggregate types. One is for projects, the second one is for team members and the third is for teams. The existing event definitions lack multiple items in order to create a complete set of state changes. This circumstance underlines the difference between Domain Events and Event Sourcing records. Other than the user part, the project context requires to forward one type to the Event Bus as Domain Event. The resulting notifications are consumed both internally and externally by the task board context.

The first code shows the extended event definitions:

Project context: Events

```
const ProjectCreatedEvent = createEventType('ProjectCreated', {projectId: 'string',
  name: 'string', ownerId: 'string', teamId: 'string', taskBoardId: 'string'});
const ProjectRenamedEvent = createEventType(
  'ProjectRenamed', {projectId: 'string', name: 'string'});
const TeamMemberCreatedEvent = createEventType(
  'TeamMemberCreated', {teamMemberId: 'string', userId: 'string', role: 'string'});
const TeamMemberRoleChangedEvent = createEventType(
  'TeamMemberRoleChanged', {teamMemberId: 'string', role: 'string'});
const TeamCreatedEvent = createEventType('TeamCreated', {teamId: 'string'});
const TeamMemberAddedToTeamEvent = createEventType(
  'TeamMemberAddedToTeam', {teamId: 'string', teamMemberId: 'string'});
const TeamMemberRemovedFromTeamEvent = createEventType(
  'TeamMemberRemovedFromTeam', {teamId: 'string', teamMemberId: 'string'});
```

The next three examples show the refactored Write Model components for projects, teams and team members:

Project context: Project Write Model

```
const applyProjectEvents = (state, events) => events.reduce(applyEvent, state);

const applyEvent = (state, event) => event.type === ProjectCreatedEvent.type ?
  {...state, id: event.data.projectId} : {...state};

const createProject = ({id: projectId, name, ownerId, teamId, taskBoardId}) => {
  verify('valid data', projectId && name && ownerId && teamId && taskBoardId);
  verify('valid name', typeof name === 'string' && !!name);
  return [new ProjectCreatedEvent(
    {projectId, name, ownerId, teamId, taskBoardId})];
};

const renameProject = (state, name) => {
  verify('valid name', typeof name === 'string' && !!name);
  return [new ProjectRenamedEvent({projectId: state.id, name})];
};
```

Project context: Team member Write Model

```
const applyTeamMemberEvents = (state, events) => events.reduce(applyEvent, state);

const applyEvent = (state, {data, type}) => {
  if (type === TeamMemberCreatedEvent.type)
    return {...state,
      id: data.teamMemberId, userId: data.userId, role: new Role(data.role)};
  return {...state};
};

const createTeamMember = ({id, userId, role}) => {
  verify('valid id', id != null);
  verify('valid user id', userId != null);
  return [new TeamMemberCreatedEvent({teamMemberId: id, userId, role: role.name})];
};

const updateTeamMemberRole = (state, role) => {
  verify('valid role', role.constructor === Role);
  return [new TeamMemberRoleChangedEvent(
    {teamMemberId: state.id, role: role.name})];
};
```

Project context: Team Write Model

```
const applyTeamEvents = (state, events) => events.reduce(applyEvent, state);

const applyEvent = (state, event) => {
  const {data} = event;
  if (event.type === TeamCreatedEvent.type)
    return {...state, id: data.teamId, teamMemberIds: []};
  if (event.type === TeamMemberAddedToTeamEvent.type)
    return {...state,
      teamMemberIds: (state.teamMemberIds || []).concat(data.teamMemberId)};
  if (event.type === TeamMemberRemovedFromTeamEvent.type)
    return {...state, teamMemberIds:
      (state.teamMemberIds || []).filter(id => id !== data.teamMemberId)};
  return {...state};
};

const createTeam = id => {
  verify('valid id', id != null);
  return [new TeamCreatedEvent({teamId: id})];
```

```
};

const addTeamMember = (state, teamMemberId) => {
  verify('team member is new', !state.teamMemberIds.includes(teamMemberId));
  return [new TeamMemberAddedToTeamEvent({teamId: state.id, teamMemberId})];
};

const removeTeamMember = (state, teamMemberId) => {
  const indexToRemove = state.teamMemberIds.indexOf(teamMemberId);
  if (indexToRemove === -1) return [];
  return [new TeamMemberRemovedFromTeamEvent({teamId: state.id, teamMemberId})];
};
```

The event definitions are complemented with the types `TeamMemberRoleChangedEvent` and `TeamCreatedEvent`. This creates a complete collection of all state transitions within the conceptual boundary. The three Aggregate types for projects, team members and teams are refactored to event-sourced components using a functional approach. Each of them provides a dedicated operation for building a state representation. In terms of behavior, every component has one function for creating a new instance and others to modify existing ones. Again, the resulting code is significantly shorter compared to the previous Write Model implementations of the project context. The project Aggregate type is one example for the structural freedom with event-sourced Write Models. While a project encloses many individual fields, its state representation exclusively incorporates an identifier.

The next code illustrates the refactored Command Handlers component:

Project context: Command Handlers

```
class ProjectCommandHandlers {

  #eventStore;

  constructor({eventStore}) {
    this.#eventStore = eventStore;
  }

  handleCommand = createMessageForwarder(this, {messageSuffix: 'Command'});

  async handleCreateProjectCommand({data}) {
    const {projectId: id, name, ownerId, teamId, taskBoardId} = data;
    const events = createProject({id, name, ownerId, teamId, taskBoardId});
    await this.#eventStore.save(
      `project/${id}`, events, {expectedVersion: 0});
  }
}
```

```
}

async handleUpdateProjectNameCommand({data: {projectId: id, name}}) {
  const {events, currentVersion} = await this.#eventStore.load(`project/${id}`);
  const newEvents = renameProject(applyProjectEvents({}, events), name);
  await this.#eventStore.save(
    `project/${id}`, newEvents, {expectedVersion: currentVersion});
}

async handleAddTeamMemberToTeamCommand(command) {
  const {data: {teamId, teamMemberId, userId, role}} = command;
  const teamMemberEvents = createTeamMember(
    {id: teamMemberId, userId, role: new Role(role)});
  await this.#eventStore.save(`team-member/${teamMemberId}`, teamMemberEvents);
  const {events, currentVersion} = await this.#eventStore.load(`team/${teamId}`);
  const newTeamEvents = addTeamMember(applyTeamEvents({}, events), teamMemberId);
  await this.#eventStore.save(
    `team/${teamId}`, newTeamEvents, {expectedVersion: currentVersion});
}

/*
 * .. handleRemoveTeamMemberFromTeamCommand() .. */
/*
 * .. handleUpdateTeamMemberRoleCommand() .. */

}
```

The class `ProjectCommandHandlers` is reworked to use an Event Store for persistence and event-sourced components for behavior execution. Overall, the pattern for individual operations is similar to the Command Handlers of the user context. One notable exception is the function `handleAddTeamMemberToTeamCommand()`, which affects multiple Aggregate instances and therefore spans across transactions. In terms of consistency and error-proneness, this approach is identical to the previous chapter. While an execution failure can result in an orphaned team member Aggregate, the implementation does not risk producing corrupt states.

As with the user context, other components of the Application Layer in the project context are also adjusted. The class `ProjectDomainEventHandlers` is changed to work with an event-sourced team Aggregate type. However, the subscription to the Event Bus remains, as the creation of a project is considered a Domain Event. For the read part, the synchronization class `ProjectReadModelSynchronization` is converted to a projection component and named `ProjectReadModelProjection`. The Query Handlers of the project part require no changes.

The following example shows a usage of the project context with Event Sourcing and

separated Domain Event publishing ([run code](#)):

Project context: Overall usage

```
const projectCommandHandlers = new ProjectCommandHandlers({eventStore});
const projectDomainEventHandlers = new ProjectDomainEventHandlers(
  {eventStore, eventBus});
projectDomainEventHandlers.activate();
const projectReadModelStorage =
  new InMemoryIndexedStorage({indexes: ['teamId', 'ownerId']});
const teamMemberReadModelStorage = new InMemoryIndexedStorage({indexes: ['userId']});
const projectReadModelProjection = new ProjectReadModelProjection(
  {projectReadModelStorage, teamMemberReadModelStorage, eventStore});
projectReadModelProjection.activate();
const projectQueryHandlers = new ProjectQueryHandlers(
  {projectReadModelStorage, teamMemberReadModelStorage});

const teamId = generateId(), teamMemberId = generateId(), userId = generateId();
const projectId = generateId(), taskBoardId = generateId();

projectCommandHandlers.handleCommand(new CreateProjectCommand(
  {data: {projectId, name: 'Test', ownerId: userId, teamId, taskBoardId}}));
await timeout(125);
projectCommandHandlers.handleCommand(new AddTeamMemberToTeamCommand(
  {data: {teamId, teamMemberId, userId, role: 'developer'}}));
projectCommandHandlers.handleCommand(new UpdateProjectNameCommand(
  {data: {projectId, name: 'Test Project v2'}}));
await timeout(125);
console.log(await projectQueryHandlers.handleQuery(
  new FindProjectsByOwnerQuery({data: {userId}})));
console.log(await projectQueryHandlers.handleQuery(
  new FindProjectsByCollaboratingUserQuery({data: {userId}})));
```

Task board context

In terms of required modifications, the task board part represents a combination of the other two contexts. For one, it also contains multiple Aggregate types that need to be refactored into an event-sourced implementation. The existing event definitions again only miss a single item in order to represent a full collection of state transitions. Also, the context incorporates a Domain Event handlers component on the write side. Similar to the user context, this part maintains a specialized Read Model that requires an according projection mechanism. With

regard to event publishing, the task board part both produces and processes Domain Events. While one of the processed types is from inside the context, the other one originates from the project part.

The first code shows the full event definitions for the task board context:

Task board context: Events

```
const TaskCreatedEvent = createEventType('TaskCreated',
  {taskId: 'string', title: 'string', description: 'string',
   status: ['string', 'undefined'], assigneeId: ['string', 'undefined']}));
const TaskTitleChangedEvent = createEventType(
  'TaskTitleChanged', {taskId: 'string', title: 'string'});
const TaskDescriptionChangedEvent = createEventType(
  'TaskDescriptionChanged', {taskId: 'string', description: 'string'});
const TaskAssigneeChangedEvent = createEventType(
  'TaskAssigneeChanged', {taskId: 'string', assigneeId: ['string', 'undefined']}));
const TaskStatusChangedEvent = createEventType(
  'TaskStatusChanged', {taskId: 'string', status: 'string'});
const TaskBoardCreatedEvent = createEventType(
  'TaskBoardCreated', {taskBoardId: 'string'});
const TaskAddedToTaskBoardEvent = createEventType(
  'TaskAddedToTaskBoard', {taskBoardId: 'string', taskId: 'string'});
const TaskRemovedFromTaskBoardEvent = createEventType(
  'TaskRemovedFromTaskBoard', {taskBoardId: 'string', taskId: 'string'});
```

The following two examples provide the reworked Aggregate types for task and task board:

Task board context: Task Write Model

```
const validStatus = ['todo', 'in progress', 'done'];

const applyTaskEvents = (state, events) => events.reduce(applyEvent, state);

const applyEvent = (state, event) => {
  const {data} = event;
  if (event.type === TaskCreatedEvent.type)
    return {...state, id: data.taskId, assigneeId: data.assigneeId};
  if (event.type === TaskStatusChangedEvent.type)
    return {...state, status: data.status};
  if (event.type === TaskAssigneeChangedEvent.type)
    return {...state, assigneeId: data.assigneeId};
  return {...state};
};
```

```

const createTask = ({id, title, description, status = 'todo', assigneeId}) => {
  verify('valid id', id != null);
  verify('valid title', typeof title === 'string' && !!title);
  verify('valid status', validStatus.includes(status));
  verify('active task assignee', status !== 'in progress' || assigneeId);
  return [new TaskCreatedEvent(
    {taskId: id, title, description, status, assigneeId})];
};

const updateTitle = (state, title) => {
  verify('valid title', typeof title === 'string' && !!title);
  return [new TaskTitleChangedEvent({taskId: state.id, title})];
};

const updateDescription = (state, description) =>
  [new TaskDescriptionChangedEvent({taskId: state.id, description})];

const updateStatus = (state, status) => {
  verify('valid status', validStatus.includes(status));
  verify('active task assignee', status !== 'in progress' || state.assigneeId);
  return [new TaskStatusChangedEvent({taskId: state.id, status})];
};

const updateAssignee = (state, assigneeId) => {
  verify('active task assignee', state.status !== 'in progress' || assigneeId);
  return [new TaskAssigneeChangedEvent({taskId: state.id, assigneeId})];
};

```

Task board context: Task Board Write Model

```

const applyTaskBoardEvents = (state, events) => events.reduce(applyEvent, state);

const applyEvent = (state, event) => {
  const {data} = event;
  if (event.type === TaskBoardCreatedEvent.type)
    return {...state, id: data.taskBoardId, taskIds: []};
  if (event.type === TaskAddedToTaskBoardEvent.type)
    return {...state, taskIds: (state.taskIds || []).concat(data.taskId)};
  if (event.type === TaskRemovedFromTaskBoardEvent.type)
    return {...state,
      taskIds: (state.taskIds || []).filter(id => id !== data.taskId)};
  return {...state};
}

```

```
};

const createTaskBoard = id => {
  verify('valid id', id != null);
  return [new TaskBoardCreatedEvent({taskBoardId: id})];
};

const addTask = (state, taskId) => {
  verify('valid task id', taskId != null);
  return [new TaskAddedToTaskBoardEvent({taskBoardId: state.id, taskId})];
};

const removeTask = (state, taskId) => {
  verify('task is on board', state.taskIds.indexOf(taskId) > -1);
  return [new TaskRemovedFromTaskBoardEvent({taskBoardId: state.id, taskId})];
};
```

In addition to the existing event definitions, the type `TaskBoardCreatedEvent` is introduced for a complete collection of state changes. The two Aggregate types for task and task board are refactored to functional event-sourced variants. Their implementation is very similar to the components of the other context implementation. One operation is for building a state representation, another function creates a new instance, and others update existing instances. The task operation `updateDescription()` is an example of a minimal implementation for a behavior using a functional approach. As the action exclusively returns an event, it can even be implemented with a concise function body. With an object-oriented approach, this would likely not be possible due to necessity of maintaining an internal state representation.

The next code shows the Command Handlers class for the event-sourced implementation:

Task board context: Command Handlers

```
class TaskBoardCommandHandlers {

  #eventStore;

  constructor({eventStore}) {
    this.#eventStore = eventStore;
  }

  handleCommand = createMessageForwarder(this, {messageSuffix: 'Command'});

  async handleAddNewTaskToTaskBoardCommand({data}) {
    const {taskId, taskBoardId, title, description, status, assigneeId} = data;
```

```
const events = createTask({id: taskId, title, description, status, assigneeId});
await this.#eventStore.save(`task/${taskId}`, events, {expectedVersion: 0});
const {events: taskBoardEvents, currentVersion} =
  await this.#eventStore.load(`task-board/${taskBoardId}`);
const newTaskBoardEvents = addTask(
  applyTaskBoardEvents({}, taskBoardEvents), taskId);
await this.#eventStore.save(`task-board/${taskBoardId}`,
  newTaskBoardEvents, {expectedVersion: currentVersion});
}

async handleUpdateTaskTitleCommand({data: {taskId, title}}) {
  const {events, currentVersion} = await this.#eventStore.load(`task/${taskId}`);
  const newEvents = updateTitle(applyTaskEvents({}, events), title);
  await this.#eventStore.save(
    `task/${taskId}`, newEvents, {expectedVersion: currentVersion});
}

async handleUpdateTaskDescriptionCommand({data: {taskId, description}}) {
  const {events, currentVersion} = await this.#eventStore.load(`task/${taskId}`);
  const newEvents = updateDescription(applyTaskEvents({}, events), description);
  await this.#eventStore.save(
    `task/${taskId}`, newEvents, {expectedVersion: currentVersion});
}

async handleRemoveTaskFromTaskBoardCommand({data: {taskBoardId, taskId}}) {
  const {events, currentVersion} =
    await this.#eventStore.load(`task-board/${taskBoardId}`);
  const newEvents = removeTask(applyTaskBoardEvents({}, events), taskId);
  await this.#eventStore.save(
    `task-board/${taskBoardId}`, newEvents, {expectedVersion: currentVersion});
}

/* .. handleUpdateTaskDescriptionCommand() .. */
/* .. handleUpdateTaskAssigneeCommand() .. */
/* .. handleUpdateTaskStatusCommand() .. */

}
```

This is followed by a refactored Domain Event handlers component:

Task context: Domain Event Handlers

```

class TaskBoardDomainEventHandlers {

  constructor({eventStore, eventBus, taskAssigneeReadModelStorage}) {
    this.activate = () => {
      eventStore.subscribe('$global', ({type, data}) => {
        if ([TaskCreatedEvent.type, TaskAssigneeChangedEvent.type].includes(type))
          taskAssigneeReadModelStorage.update(data.taskId,
            {id: data.taskId, assigneeId: data.assigneeId});
      });
      eventBus.subscribe('ProjectCreated', async ({data: {taskBoardId: id}}) => {
        await eventStore.save(
          `task-board/${id}`, createTaskBoard(id), {expectedVersion: 0});
      });
      eventBus.subscribe('TeamMemberRemovedFromTeam', async ({data}) => {
        const taskAssignees = taskAssigneeReadModelStorage.findIndex(
          'assigneeId', data.teamMemberId);
        await Promise.all(taskAssignees.map(async ({id}) => {
          const {events, currentVersion} = await eventStore.load(`task/${id}`);
          const taskState = applyTaskEvents({}, events);
          const statusEvents = taskState.status === 'in progress' ?
            updateStatus(taskState, 'todo') : [];
          const updatedTaskState = applyTaskEvents(taskState, statusEvents);
          const assigneeEvents = updateAssignee(updatedTaskState, undefined);
          await eventStore.save(`task/${id}`, [...statusEvents, ...assigneeEvents],
            {expectedVersion: currentVersion});
        }));
      });
    };
  };
}

```

The class `TaskBoardCommandHandlers` is changed similarly to the counterparts of the other context implementations. Its operation `handleAddNewTaskToTaskBoardCommand()` is another example for an event-sourced use case that affects multiple transactions. The component `TaskBoardDomainEventHandlers` is changed in different ways. For maintaining the task assignee Read Models, it consumes events from the Event Store instead of listening to the Event Bus. The other Event Bus subscribers for the event types “`ProjectCreated`” and “`TeamMemberRemovedFromTeam`” remain active. However, their callback implementations are adjusted to work with event-sourced Write Model components. Furthermore, un-assigning

tasks from a removed team member is an example for executing multiple actions on the same event-sourced component. The class `TaskBoardReadModelSynchronization` is transformed into the projection component `TaskBoardReadModelProjection`. Again, the Query Handlers remain unchanged.

The usage of the event-sourced project context can be seen in the following example ([run code](#)):

Task board context: Overall usage

```
const taskBoardCommandHandlers = new TaskBoardCommandHandlers(
  {eventStore});
const taskBoardDomainEventHandlers = new TaskBoardDomainEventHandlers(
  {eventStore, eventBus, taskAssigneeReadModelStorage});
taskBoardDomainEventHandlers.activate();

const taskReadModelStorage = new InMemoryIndexedStorage({indexes: ['taskBoardId']});
const taskBoardQueryHandlers = new TaskBoardQueryHandlers({taskReadModelStorage});
const taskBoardReadModelProjection = new TaskBoardReadModelProjection(
  {eventStore, taskReadModelStorage});
taskBoardReadModelProjection.activate();

const taskBoardId = generateId(), taskId = generateId(), assigneeId = generateId();

await eventBus.publish({type: 'ProjectCreated', data: {taskBoardId}});
await taskBoardCommandHandlers.handleCommand(new AddNewTaskToTaskBoardCommand(
  {data: {taskId, taskBoardId, title: 'tests', description: 'write unit tests'}}));
await taskBoardCommandHandlers.handleCommand(
  new UpdateTaskAssigneeCommand({data: {taskId, assigneeId}}));
await taskBoardCommandHandlers.handleCommand(
  new UpdateTaskStatusCommand({data: {taskId, status: 'in progress'}}));
await timeout(125);
await eventBus.publish(new TeamMemberRemovedFromTeamEvent(
  {teamId: generateId(), teamMemberId: assigneeId}));
await timeout(125);
console.log(await taskBoardQueryHandlers.handleQuery(
  new FindTasksOnTaskBoardQuery({data: {taskBoardId}})));
```

The refactored Sample Application implementation with Event Sourcing provides multiple advantages. For one, the complexity of the Domain Layer is reduced. This is because the functional implementations contain less code and the factories are removed completely. Secondly, state transitions become first class citizens and are clearly differentiated from

Domain Events. Also, continuous projections eliminate the need of a Read Model rebuild mechanism. The next step is to break up the software into separate executable programs.

Chapter 13: Separate executable programs

Most software that consists of multiple parts, where each deals with specialized aspects, can be divided into separate executable programs. This separation enables to operate autonomous and self-contained runtime units that ideally do not influence each other's performance and availability. Also, it allows the infrastructure for each program to scale independently. For every information exchange between separate programs, an according inter-process communication must be established. Furthermore, the introduction of a technological boundary demands to define a binding contract. Failing to separate the execution of autonomous software parts risks to make their domain-specific functionalities interdependent. At the same time, an artificial division into individual programs can introduce unnecessary complexity. Especially small projects are better operated as single monolithic executable, regardless of separation capabilities.



What about Microservices?

The following sections explain concepts and mechanisms that can produce a similar result as when employing a Microservices-based architecture. This chapter remains free of such a particular architectural style, as it is not within the scope of the book.

Program layouts

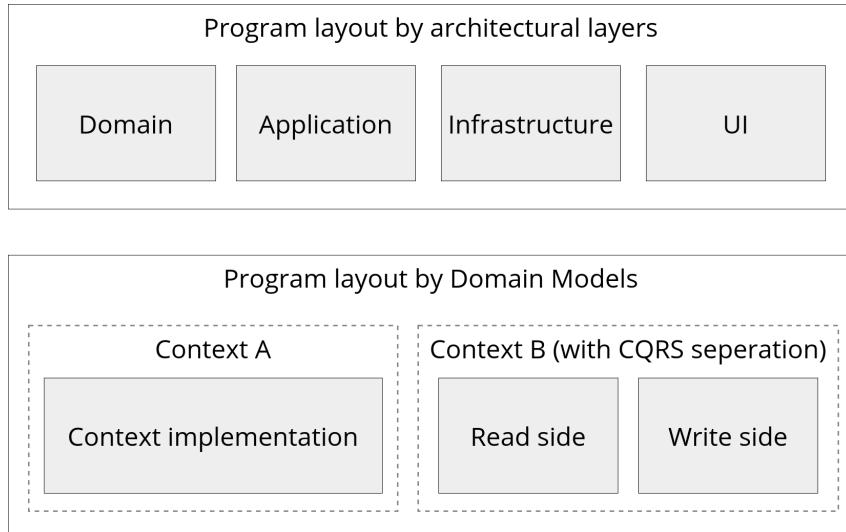


Figure 13.1: Program layouts

There are different layout possibilities for dividing a software into executable programs. One option is to separate architectural layers. For example, the Domain part can be operated independently of others. Another possibility is to align with conceptual boundaries. The implementations for individual contexts can be operated as standalone programs. This produces a one-to-one alignment of Domain Models and runtime units. When applying CQRS, the write side and the read side of an implementation should also be separated. With regard to Event Sourcing, the Event Store must support inter-process access. Furthermore, a program layout can also mix multiple approaches. Independent of layouts, one option for scaling is to operate multiple processes of the same part. However, this requires infrastructural components to support inter-process concurrent usage.

Terms related to programs and processes

Term	Meaning
Software	Combination of programs and data
Program	Collection of instructions
Process	Execution of a program
Thread	Component of a process



One process per functionality

The examples and the Sample Application implementation in this chapter operate at most one Node.js process per individual functionality. While scaling software parts beyond a single process is an important aspect, it is not the focus of this chapter. Enabling the exemplary filesystem-based infrastructure components to correctly work across threads would introduce unnecessary complexity.

Context-level communication

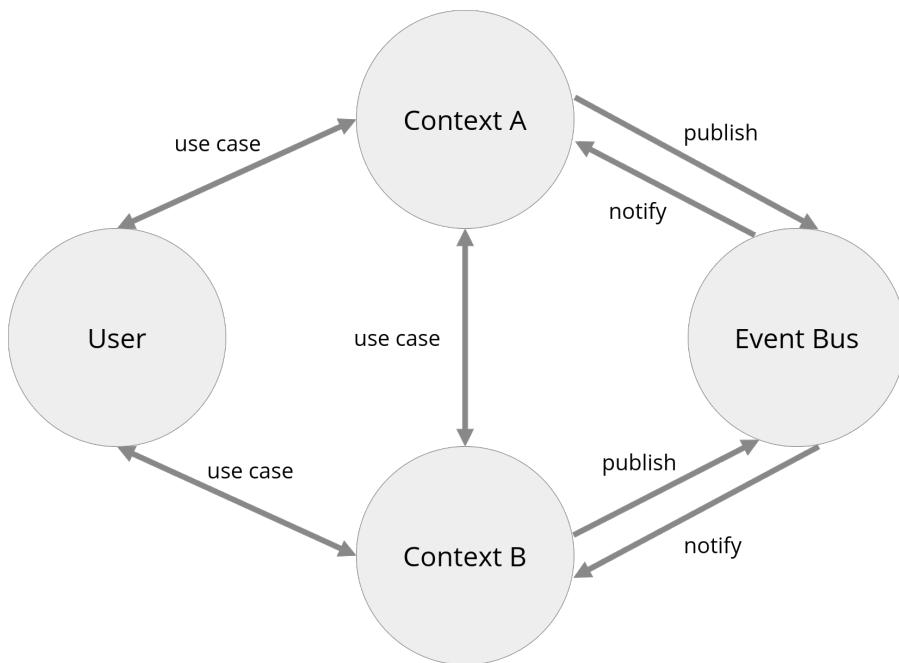


Figure 13.2: Context-level communication

For software that applies DDD and CQRS, there are two types of context-level communication. The first one is to request a use case execution by issuing a Command or a Query. This procedure requires a bidirectional one-to-one information exchange, during which both parties must remain available. The requester provides inputs and awaits a response, either in form of data or an operational status. Most commonly, a remote use case execution is issued by a user. The second communication type is the notification about an occurrence by distributing a Domain Event via an Event Bus. This mechanism establishes a unidirectional

information flow with one sender and many receivers. The producer of a message exclusively awaits acknowledgment from the Event Bus, independent of the actual consumers.



Relation to IPC

Inter-process Communication (IPC) is the exchange of information across processes. Every Operating System provides various mechanisms for this purpose. This includes communication via the filesystem, through sockets or with shared memory. The approaches differ in aspects such as whether they are unidirectional or bidirectional, the number of recipients and their reliability. On top of native mechanisms, custom protocols can be implemented.

Remote use case execution

The execution of a use case that belongs to a foreign program requires an ad-hoc bidirectional inter-process communication mechanism. Typically, a two-way connection is established on demand, the according information is exchanged and the connection is discarded. For performance reasons, past connections can also be retained or re-used. During the information exchange, both involved parties must remain available. The requester starts with sending a message that contains a type and associated inputs. On the receiver side, the provided inputs are processed, the target action is executed and the result is returned. When a responder is unavailable, either before a connection attempt or during a transfer, the requester must take according action. This can include a retry behavior, a specialized fallback mechanism or an exception reporting.



Queues for Commands and Queries

Commands and Queries can be queued and be processed asynchronously instead of directly executing handler operations. This approach can increase resilience through message relaying and load distribution. However, it lacks a communication back-channel for response data and operational statuses. In fact, messages inside a queue may conceptually be really events, as they cannot be rejected, but only acknowledged.

HTTP interface

The Hypertext Transfer Protocol (HTTP) is a commonly used inter-process communication mechanism for remote use case execution. There are multiple reasons for this. For one, it is very widespread due to the prevalence of the Internet. Secondly, it is supported by browsers, which allows the same communication protocol between software parts and with the User Interface. The HTTP defines request methods to represent different types of actions and response status codes to indicate operational success or failure. For Commands, the method POST can be used and the custom data can be sent as request body. For Queries, the verb GET is most appropriate and the custom data can be transferred as query string. However, this requires a specialized serialization mechanism to support nested object structures.

Relevant components of HTTP

Term	Meaning
Request method	Type of action to perform (alias: verb)
Request body	Custom data of a request (optional)
Response status code	Indication of operational success/failure
URL Query string	Collection of custom parameters and values

As preparation for an HTTP interface, the following examples provide functions to serialize and de-serialize an object as query string:

HTTP: Create query string helper function

```
const createQueryString = queryParameters =>
  Object.entries(queryParameters).map(([key, value]) =>
    typeof value === 'object' ?
      Object.entries(value).map(([subKey, value]) => `${key}.${subKey}=${value}`) :
      `${key}=${value}`)
    .flat(Infinity).join('&');
```

HTTP: Parse query string helper function

```
const parseQueryString = url => {
  const parsedQuery = parse(url, true).query;
  const parsedQueryWithNesting = {};
  Object.entries(parsedQuery).forEach(([key, value]) => {
    const [key, nestedKey] = key.split('.');
    if (!nestedKey) parsedQueryWithNesting[key] = value;
    else {
      parsedQueryWithNesting[key] = parsedQueryWithNesting[key] || {};
      parsedQueryWithNesting[key][nestedKey] = value;
    }
  });
  return parsedQueryWithNesting;
};
```

This is complemented with an operation to parse an HTTP request body stream:

HTTP: Parse HTTP stream helper function

```
const parseHttpStream = stream =>
  new Promise(resolve => {
    let result = '';
    stream.on('data', chunk => result += chunk.toString());
    stream.on('end', () => resolve(result));
  });
};
```

The function `createQueryString()` converts an object into a query string. As first step, it iterates the key-value pairs of a given structure and transforms each entry into a string. If a value is an object itself, all nested properties are serialized separately with the parent property as prefix. Afterwards, the list of strings is combined and returned. The operation `parseQueryString()` de-serializes a query string. This is done by executing the Node.js function `url.parse()` and further transforming the result for grouping nested properties. Note that both operations only support one level of nesting. The function `parseHttpStream()` parses an HTTP request body by registering according event handlers to accumulate the incoming message data. The operation returns a Promise instance that resolves with the final result.

The following code provides a factory to create an HTTP interface:

HTTP: HTTP interface factory

```
const createHttpInterface = (targetFunction, methods = []) =>
  async (request, response) => {
    try {
      if (!methods.includes(request.method)) throw new Error('invalid method');
      const message = request.method === 'POST' ?
        JSON.parse(await parseHttpStream(request)) : parseQueryString(request.url);
      const result = await targetFunction(message);
      response.writeHead(200,
        {'Cache-Control': 'no-cache', 'Content-Type': 'application/json'});
      response.end(result != null ? JSON.stringify(result) : result);
    } catch (error) {
      response.writeHead(500);
      response.end(error.toString());
    }
  };

```

The factory `createHttpInterface()` creates an HTTP interface for a given function. As arguments, it expects a target operation and allowed methods. Its return value is a request listener function for a Node.js Server instance. As first step, the operation checks the request method and throws an error for disallowed values. Next, it retrieves the actual message. For POST requests, the body is parsed with the function `parseHttpStream()`. For GET requests, the query string is de-serialized via the operation `parseQueryString()`. Afterwards, the target function is invoked with the received message as argument. Finally, the headers “Cache-Control” and “Content-Type” are set and the return value is serialized and transmitted. Errors during the process produce a response with a status code of 500 together with an according message.

As preparation for a usage example, the next code provides helper operations to issue POST and GET requests:

HTTP: Request helper functions

```
const post = (url, data) => new Promise(resolve => {
  http.request(urlWithProtocol(url), {method: 'POST'}, response => {
    parseHttpStream(response).then(resolve);
  }).end(JSON.stringify(data));
});

const get = (url, queryParameters = {}) => new Promise(resolve => {
  const queryString = createQueryString(queryParameters);
  http.get(` ${urlWithProtocol(url)}?${queryString}`, response =>
    parseHttpStream(response).then(resolve));
});

const urlWithProtocol = url => url.indexOf('http') === 0 ? url : `http:// ${url}`;
```

The last example shows an exemplary usage the HTTP interface factory function ([run code](#)):

HTTP: Create HTTP interface factory usage

```
const handleCommand = async command => console.log(command);
const handleQuery = async query => query;

const commandHandlersHttpInterface = createHttpInterface(handleCommand, ['POST']);
const queryHandlersHttpInterface = createHttpInterface(handleQuery, ['GET']);

http.createServer(commandHandlersHttpInterface).listen(50000);
http.createServer(queryHandlersHttpInterface).listen(50001);

console.log(
  await post('http://localhost:50000/', {type: 'ChangeData', data: {foo: 'bar'}}));
console.log(
  await get('http://localhost:50001/', {type: 'GetData', data: {id: '123'}}));
```

The function `post()` is responsible for issuing a POST request with custom data. Sending a GET request with custom query parameters is done via the operation `get()`. The example usage starts with defining placeholder handler functions for Commands and Queries. While the operation `handleCommand()` logs every received message to the console, the function `handleQuery()` simply returns back each Query. As next step, two HTTP interfaces are created with the factory `createHttpInterface()`. Then, two HTTP servers are instantiated, the listener operations are passed in and the servers are started. The actual use cases consists of sending one POST request and one GET request, each to the according server. Executing

the code demonstrates that the HTTP interfaces correctly invoke the target function and forward its return value.



What about context-to-context communication?

The execution of a use case is typically performed in response to a request from the User Interface. While there are situations where a context requests a remote use case execution, this relationship type can often be avoided. The more scalable and resilient integration approach is to establish a communication via an Event Bus or a message queue.

Remote event distribution

The distribution of Domain Events across separate programs requires a unidirectional one-to-many inter-process communication mechanism. Typically, every subscriber registers once at the start of its execution and maintains a long-lasting connection, regardless of individual notifications. Depending on the implementation, each publisher either also maintains a permanent connection or uses an ad-hoc mechanism. Whenever a Domain Event occurs, its originating process sends an according message to the Event Bus and awaits acknowledgment. If the Event Bus is unavailable, the publisher must retry the delivery until it succeeds. Each received message causes all interested subscribers to be notified asynchronously. The sender does not need to be aware of the receivers or their availability. Even more, the Event Bus can retain messages for consumers that are temporarily offline.

The following example shows a filesystem-based Message Bus component:

Message Bus: Filesystem Message Bus

```
class FilesystemMessageBus {  
  
    #inboxDirectory; #processedIdsDirectory; #subscribersByTopic; #processingQueue;  
  
    constructor({storageDirectory, subscriberGroup = 'default'}) {  
        this.#inboxDirectory = `${storageDirectory}/_inbox/`;  
        this.#processedIdsDirectory = `${storageDirectory}/${subscriberGroup}`;  
        this.#subscribersByTopic = new Map();  
        this.#processingQueue = new AsyncQueue();  
        mkdirSync(this.#inboxDirectory, {recursive: true});  
        mkdirSync(this.#processedIdsDirectory, {recursive: true});  
        watch(this.#inboxDirectory, async (_, filename) => {  
            // Process the file  
        });  
    }  
}
```

```
    const {name: messageId, ext: extension} = path.parse(filename);
    if (extension === '.json') this.#processMessage(messageId);
  });
}

subscribe(topic, subscriber) {
  const newSubscribers = this.#getSubscribers(topic).concat([subscriber]);
  this.#subscribersByTopic.set(topic, newSubscribers);
}

unsubscribe(topic, subscriber) {
  const subscribers = this.#getSubscribers(topic);
  subscribers.splice(subscribers.indexOf(subscriber), 1);
  this.#subscribersByTopic.set(topic, subscribers);
}

async publish(topic, message) {
  await writeFileAtomically(`.${this.#inboxDirectory}/${message.id}.json`,
    JSON.stringify({topic, message}));
}

#processMessage(messageId) {
  this.#processingQueue.enqueueOperation(async () => {
    const isNewItem = await access(`.${this.#processedIdsDirectory}/${messageId}`)
      .then(() => false).catch(() => true);
    if (!isNewItem) return;
    const {topic, message} = JSON.parse(await readFile(
      `.${this.#inboxDirectory}/${messageId}.json`, 'utf-8'));
    await Promise.all(this.#getSubscribers(topic).map(subscriber =>
      new Promise(resolve => setTimeout(() => {
        Promise.resolve(subscriber(message)).then(resolve);
      })),
    ));
    await writeFile(`.${this.#processedIdsDirectory}/${messageId}`, '');
  });
}

#getSubscribers(topic) { return this.#subscribersByTopic.get(topic) || []; }
```

The class `FilesystemMessageBus` implements a filesystem-based Message Bus. Each published item is saved in an inbox directory and its processing is tracked per subscriber group. The

component expects a storage directory and a group identifier. Its constructor creates required directories, instantiates the `AsyncQueue` class and initializes a filesystem watcher. Both the functions `subscribe()` and `unsubscribe()` are identical to the in-memory implementation. The command `publish()` persists a message in the inbox directory. This activity triggers the callback `#processMessage()` across individual programs. The operation enqueues a notification process. First, the received message is verified to be new. Then, the data is loaded and interested subscribers are notified. Finally, the message is marked as processed for the respective group. Again, the implementation ensures an “at least once” delivery.

The next examples provide the code for two exemplary processes that communicate via the Message Bus component ([run code](#)):

Message Bus: Filesystem Message Bus usage

```
const messageBus = new FilesystemMessageBus(  
  {storageDirectory, subscriberGroup: 'a'});  
const eventBus = new EventBus(messageBus);  
  
eventBus.subscribe('MessageFromProcessB',  
  event => console.log('received in process a', event));  
  
setTimeout(() =>  
  eventBus.publish({id: generateId(), type: 'MessageFromProcessA'}), 125);
```

Message Bus: Filesystem Message Bus usage

```
const messageBus = new FilesystemMessageBus(  
  {storageDirectory, subscriberGroup: 'b'});  
const eventBus = new EventBus(messageBus);  
  
eventBus.subscribe('MessageFromProcessA',  
  event => console.log('received in process b', event));  
  
setTimeout(() =>  
  eventBus.publish({id: generateId(), type: 'MessageFromProcessB'}), 125);
```

The code examples start with creating an instance of the class `FilesystemMessageBus`. For that, the first process defines the subscriber group “a”, while the second one uses the value “b”. Afterwards, the class `EventBus` is instantiated with the filesystem Message Bus as constructor argument. Next, a subscriber operation is registered at the Event Bus for receiving a message from the other process. The callback function simply logs each received

event to the console. Finally, the operation `setTimeout()` is invoked to register a function that creates and publishes an exemplary event. When spawning both processes, one event is received in each of them and is logged to the console. The usage of the function `setTimeout()` is required to avoid lost events due to timing issues.

Sample Application: Separate executable programs

This section illustrates the separation of the Sample Application implementation into multiple executable programs. The goal is to operate autonomous and self-contained runtime parts that are independent of each other. As explained earlier, only a single Node.js process is executed per individual functionality. Every information exchange between separate parts is enabled through an according inter-process communication mechanism. The introduction of standalone programs requires to extend the implementation with further infrastructural functionalities and runtime configurations for each part. However, the existing source code of the Sample Application does not require any adaptations. Consequently, the following subsections only illustrate the additional components, the context configurations and example usages. All previously existing parts of the implementation are excluded, as they remain unmodified.

Program layout

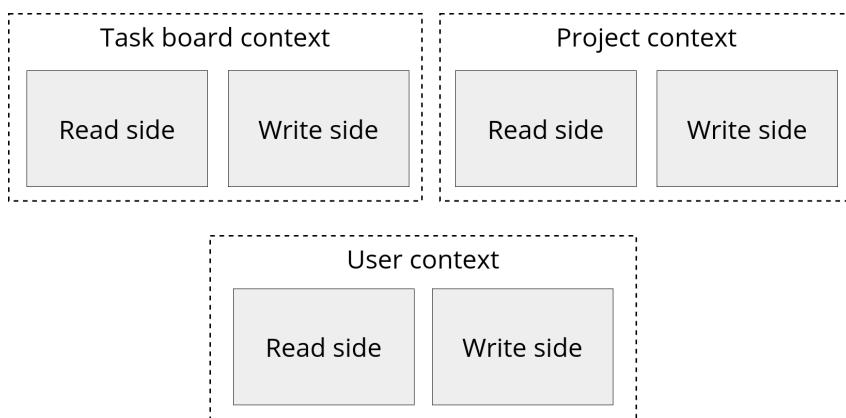


Figure 13.3: Sample Application Program layouts

The division of the Sample Application into multiple runtime parts can be done in different ways. One option is to separate the overall implementation by architectural layers. While this eventually increases performance and load capacity, it risks to make different contexts interdependent. The more useful approach is to align with the conceptual boundaries. In fact, the source code for the contexts is already separated through the existing directory structure. Furthermore, every context-overarching communication is done via an Event Bus. Consequently, the implementations can easily be operated as standalone programs. However, every part contains a write side and a read side that serve different purposes. These two should also be separated. Therefore, each of the Sample Application context implementations is split into two separate programs.

Infrastructure code

The separation into multiple executable programs requires to introduce additional generic components. For an inter-process event distribution, the class `FilesystemMessageBus` is reused. The remote execution of use cases is enabled through the factory `createHttpInterface()` and its related operations `createQueryString()`, `parseQueryString()` and `parseHttpStream()`. Also, the HTTP helper functions `get()` and `post()` are added to the Sample Application codebase. Although there is no context-to-context use case execution, the introduction of HTTP interfaces allows examples to run as separate processes. Furthermore, it serves as preparation for the User Interface Layer covered in the next chapter. In order to avoid communicating with six individual HTTP endpoints, this section introduces a proxy server factory. The component allows to create a server that routes requests based on a custom URL resolution.

The following example shows the proxy server component:

Infrastructure code: HTTP proxy factory

```
const createHttpProxy = urlResolver =>
  async (request, response) => {
    const targetUrl = await urlResolver(request);
    if (!targetUrl) {
      response.writeHead(404);
      response.end();
      return;
    }
    const forwardingRequest = http.request(targetUrl, {
      method: request.method,
      headers: request.headers,
    }, actualResponse => {
```

```
    response.writeHead(actualResponse.statusCode, actualResponse.headers);
    actualResponse.pipe(response);
  });
  request.pipe(forwardingRequest);
};
```

The function `createHttpProxy()` is a factory for creating an HTTP proxy that forwards requests based on a custom URL resolution. As only argument, it expects a callback function that maps a request object to a URL. The overall return value is an operation that can be used as listener function for an instance of `http.Server`. For every provided request, the listener first invokes the URL resolver callback. In case of a falsy return value, the connection is closed with the status code 404. Otherwise, a forwarding request is created via the operation `http.request()`. The original method and headers are re-used and the request stream is forwarded. Upon response, the incoming status code and headers are returned to the original requester together with the response stream.

The next code shows an exemplary usage of the proxy server component ([run code](#)):

Infrastructure code: Proxy server factory usage

```
http.createServer(_, response) =>
  response.end('response from the write side')).listen(50001);

http.createServer(_, response) =>
  response.end('response from the read side')).listen(50002);

const httpProxy = createHttpProxy(request => {
  const {pathname} = url.parse(request.url);
  return `http://localhost:${pathname.substr(1) === 'write-side' ? 50001 : 50002}`;
});

http.createServer(httpProxy).listen(50000);

console.log(await get('http://localhost:50000/write-side'));
console.log(await get('http://localhost:50000/read-side'));
```

Context configurations

Every Sample Application context implementation is extended with two runtime configurations. One is for the write side, the other one is for the read side. Similar to the

usage examples of the previous chapter, each configuration pair performs the necessary setup for a context. This consists of instantiating infrastructural functionalities as well as Application Layer components. For a write side, the infrastructure part typically includes an Event Store, an Event Bus and a Domain Event publisher. For each read side, it further incorporates the setup of Read Model storage mechanisms. The Application part setup consists of Application Services and optional projection components. On top of that, every configuration is complemented with the creation of an HTTP interface and an HTTP server.

The first two examples show the configuration for the write side and the read side of the user context:

User context: Write side

```
eventTypeFactory.setIdGenerator(generateId);
eventTypeFactory.setMetadataProvider(() => ({creationTime: new Date()}));

const eventStore = new ConcurrencySafeFilesystemEventStore(
  {storageDirectory: `${rootStorageDirectory}/user/event-store`});

const emailRegistry = new EmailRegistry();

const commandHandlers = new UserCommandHandlers(
  {eventStore, emailRegistry, hashPassword: createMd5Hash});
new EmailRegistryProjection({emailRegistry, eventStore}).activate();

const httpInterface = createHttpInterface(
  message => commandHandlers.handleCommand(message), ['POST']);
http.createServer(httpInterface).listen(process.env.PORT || 8080);
```

User context: Read side

```
const eventStore = new ConcurrencySafeFilesystemEventStore(
  {storageDirectory: `${rootStorageDirectory}/user/event-store`});

const userReadModelStorage =
  new InMemoryIndexedStorage({indexes: ['emailAddress']});
new UserReadModelProjection({userReadModelStorage, eventStore}).activate();
const queryHandlers = new UserQueryHandlers({userReadModelStorage});

const httpInterface = createHttpInterface(
  message => queryHandlers.handleQuery(message), ['GET']);
http.createServer(httpInterface).listen(process.env.PORT || 8080);
```

The next examples implement the configuration of the task board context:

Task board context: Write side

```
const createDefaultMetadata = () => ({creationTime: new Date()});
eventTypeFactory.setIdGenerator(generateId);
eventTypeFactory.setMetadataProvider(createDefaultMetadata);

const eventStore = new ConcurrencySafeFilesystemEventStore(
  {storageDirectory: `${rootStorageDirectory}/task-board/event-store`});

const eventBus = new EventBus(
  new FilesystemMessageBus({storageDirectory: `${rootStorageDirectory}/event-bus`,
    subscriberGroup: 'task-board'}));

const taskAssigneeReadModelStorage =
  new InMemoryIndexedStorage({indexes: ['assigneeId']});
const commandHandlers = new TaskBoardCommandHandlers({eventStore});
new TaskBoardDomainEventHandlers(
  {eventStore, eventBus, taskAssigneeReadModelStorage}).activate();

const httpInterface = createHttpInterface(
  message => commandHandlers.handleCommand(message), ['POST']);
http.createServer(httpInterface).listen(process.env.PORT || 8080);
```

Task board context: Read side

```
const eventStore = new ConcurrencySafeFilesystemEventStore(
  {storageDirectory: `${rootStorageDirectory}/task-board/event-store`});

const taskReadModelStorage = new InMemoryIndexedStorage({indexes: ['taskBoardId']});
const queryHandlers = new TaskBoardQueryHandlers({taskReadModelStorage});
new TaskBoardReadModelProjection({eventStore, taskReadModelStorage}).activate();

const httpInterface = createHttpInterface(
  message => queryHandlers.handleQuery(message), ['GET']);
http.createServer(httpInterface).listen(process.env.PORT || 8080);
```

The final two examples provide the configuration for the project context:

Project context: Write side

```

eventTypeFactory.setIdGenerator(generateId);
eventTypeFactory.setMetadataProvider(() => ({creationTime: new Date()}));

const eventStore = new ConcurrencySafeFilesystemEventStore(
  {storageDirectory: `${rootStorageDirectory}/project/event-store`});

const EventBus = new EventBus(new FilesystemMessageBus({
  storageDirectory: `${rootStorageDirectory}/event-bus`,
  subscriberGroup: 'project',
}));

const domainEventPublisher = new DomainEventPublisher({eventStore, EventBus,
  publicEventTypes: ['ProjectCreated', 'TeamMemberRemovedFromTeam'],
  storageDirectory: `${rootStorageDirectory}/project/domain-event-publisher`});
domainEventPublisher.activate();

const commandHandlers = new ProjectCommandHandlers({eventStore});
new ProjectDomainEventHandlers({eventStore, EventBus}).activate();

const httpInterface = createHttpInterface(
  message => commandHandlers.handleCommand(message), ['POST']);
http.createServer(httpInterface).listen(process.env.PORT || 8080);

```

Project context: Read side

```

const eventStore = new ConcurrencySafeFilesystemEventStore(
  {storageDirectory: `${rootStorageDirectory}/project/event-store`});

const projectReadModelStorage = new InMemoryIndexedStorage({indexes: ['teamId']});
const teamMemberReadModelStorage =
  new InMemoryIndexedStorage({indexes: ['userId']});
new ProjectReadModelProjection(
  {projectReadModelStorage, teamMemberReadModelStorage, eventStore}).activate();
const queryHandlers = new ProjectQueryHandlers(
  {projectReadModelStorage, teamMemberReadModelStorage});

const httpInterface = createHttpInterface(
  message => queryHandlers.handleQuery(message), ['GET']);
http.createServer(httpInterface).listen(process.env.PORT || 8080);

```

There are many similarities in the runtime configurations for the different context implementations. Each write side configures the identifier generator and the metadata provider

of the component `eventTypeFactory`. This is required for a correct event creation. Also, every configuration instantiates the `EventStore` class. While the write sides save to a store and read from it, the read sides exclusively subscribe for notifications. The task board part and the project part further set up an `EventBus` instance on their write side. This component internally uses the class `FilesystemMessageBus` for inter-process distribution. The project context implementation instantiates the `DomainEventPublisher` class for Domain Event distribution. Every read side sets up Read Model storages and projection components. Finally, each configuration invokes the factory `createHttpInterface()`, instantiates a server and starts it.



Storage directory and HTTP port configuration

The values for the root storage directory and the HTTP port of each configuration can be controlled via environment variables. This allows a surrounding mechanism to configure all executable parts in accordance to each other.

Proxy server

The use of a proxy server enables unified access instead of addressing each program directly via its HTTP server. This way, a consumer can communicate with the complete software through a single endpoint. Furthermore, it ensures to encapsulate the internal program layout and individual server addresses. Each part is identified through its name as URL path. On top of that, the proxy differentiates between Commands and Queries by the request method. Every POST request is routed to a write side, while GET requests are forwarded to a read side. One potential disadvantage of this approach is that the proxy server can become a single point of failure. However, this is not necessarily an issue, as production-grade components for this purpose are resilient and highly optimized.

The following example shows the bootstrapping of all individual parts as well as the setup of the proxy server:

Sample Application: Proxy server

```
const programMappings = {
  'user/write-side': 50001,
  'user/read-side': 50002,
  'project/write-side': 50003,
  'project/read-side': 50004,
  'task-board/write-side': 50005,
  'task-board/read-side': 50006,
};

Object.entries(programMappings).forEach(([program, PORT]) =>
  spawnedProcesses.push(childProcess.spawn('node',
    [path.join(rootDirectory, program)],
    {env: {...process.env, ROOT_STORAGE_DIR, PORT}, stdio: 'inherit'})));
}

const httpProxy = createHttpProxy(({method, url}) => {
  const {pathname, query} = parse(url);
  const program =
    `${pathname.substr(1)}/${method === 'POST' ? 'write' : 'read'}-side`;
  const targetPort = programMappings[program];
  return `http://localhost:${targetPort}${query}`;
});

http.createServer(httpProxy).listen(50000);
```

The configuration code first defines the object `programMappings`, which contains key-value pairs for program directories and HTTP ports. For each entry, a Node.js process is spawned via the function `child_process.spawn()` with the respective program directory as argument. Also, the environment variables for a root storage directory and an HTTP port are passed in. Next, the code creates an HTTP proxy via the function `createHttpProxy()`. The URL resolver callback first determines the target program directory using the request URL path and method. Then, the target HTTP port is retrieved and the final URL is returned. Note that the query string is also forwarded to retain any message data for GET requests. Finally, an HTTP server is instantiated and started with the proxy as listener function.

The final code shows an exemplary usage of the Sample Application ([run code](#)):

Sample Application: Overall usage

```
messageTypeFactory.setIdGenerator(generateId);
messageTypeFactory.setDefaultValueProvider(() => ({creationTime: new Date()}));

await import('../index.js');

const emailAddress = `johnathan${generateId()}@example.com`;
const userId = generateId(), teamId = generateId(), teamMemberId = generateId();
const projectId = generateId(), taskBoardId = generateId(), taskId = generateId();

await timeout(1000);

await post('localhost:50000/user', new CreateUserCommand({data:
  {userId, username: 'johnathan', emailAddress, password: 'pw1', role: 'user'}}));
await post('localhost:50000/project', new CreateProjectCommand(
  {data: {projectId, name: 'Test', ownerId: userId, teamId, taskBoardId}}));
await timeout(125);
await post('localhost:50000/project', new AddTeamMemberToTeamCommand(
  {data: {teamId, teamMemberId, userId, role: 'developer'}}));
await post('localhost:50000/task-board', new AddNewTaskToTaskBoardCommand(
  {data: {taskId, taskBoardId, title: 'tests', description: 'write unit tests'}}));
await post('localhost:50000/task-board',
  new UpdateTaskAssigneeCommand({data: {taskId, assigneeId: teamMemberId}}));
await timeout(125);

const [user] = JSON.parse(await get('localhost:50000/user',
  new FindUserByEmailAddressQuery({data: {emailAddress}})));
const [project] = JSON.parse(await get('localhost:50000/project',
  new FindProjectsByCollaboratingUserQuery({data: {userId: user.id}})));
const taskBoard = await get('localhost:50000/task-board',
  new FindTasksOnTaskBoardQuery({data: {taskBoardId: project.taskBoardId}}));
console.log(JSON.parse(taskBoard));
```

The separation of the Sample Application into six individual programs increases the overall resilience and performance. Furthermore, it ensures that the individual Domain Model implementations enclosed in their own Bounded Context do not affect each other's runtime. Even more, the write side and the read side of every context are cleanly separated from each other. The introduction of HTTP interfaces together with a unifying proxy server enables remote access to the Sample Application. The next and final step is to implement the User Interface Layer.

Chapter 14: User Interface

The User Interface enables humans and machines to interact with a software. For this purpose, it must display information, accept and process inputs, request state changes or behavior execution, and report results. In many cases, it also incorporates input validation. Typically, a User Interface renders visual elements that can be interacted with through input devices. For web-based software, this is commonly achieved with HTML, CSS and client-side JavaScript. In most cases, an according implementation consists of individual purpose-specific components that are composed with each other. The User Interface Layer does not necessarily only incorporate client-side components, but can also include server functionalities. Generally speaking, it contains everything that is necessary for clients to communicate with the Application Layer of a software.



Reduced example code

Some examples in this chapter omit important implementation parts such as model components or definitions for Events, Commands and Queries. Consequently, their source code contradicts with recommendations of other chapters. In this case, the motivation is to reduce the code amount, while at the same time provide fully functional examples.

HTTP file server

For a browser-based User Interface, a software must serve all necessary files. Typically, a user accesses a single URL as entry point to a web page and downloads an HTML document. This file contains markup that links to further resources, such as stylesheets and client-side JavaScript code. The additional assets are automatically requested, downloaded and interpreted by the browser. For this purpose, a software must operate an HTTP server that accepts requests to individual URLs and responds accordingly. Every response must provide the desired content together with a matching “Content-Type” header in order to display it correctly. When using client-side rendering, the server can stay completely agnostic of an Application Layer. Instead, the browser independently issues requests to Application components via a dedicated communication channel.

The first code shows a factory function for creating a simple HTTP filesystem interface:

HTTP: HTTP Filesystem interface factory

```
const createHttpFilesystemInterface = pathResolver =>
  async (request, response) => {
    const filePath = pathResolver(request);
    try {
      await stat(filePath);
      const fileExtension = path.extname(filePath);
      const contentType = contentTypeByExtension[fileExtension] || 'text/plain';
      response.writeHead(200, {'Content-Type': contentType});
      createReadStream(filePath).pipe(response);
    } catch (error) {
      response.writeHead(404);
      response.end();
    }
  };
};

const contentTypeByExtension = {
  '.js': 'application/javascript',
  '.html': 'text/html',
  '.css': 'text/css',
};
```

The factory `createHttpFilesystemInterface()` enables to serve static files over HTTP. As only argument, it expects a function that transforms a request object into a file path. Its overall return value is a listener callback that can be used as constructor argument for the built-in class `http.Server`. The operation first executes the provided function `pathResolver()` with the respective request object as argument. Next, it checks the existence of the resulting file path. In case of an error, the HTTP response status code is set to 404. If the file is existing, the correct content type is determined by mapping its extension with the object `contentTypeByExtension`. Finally, the according “Content-Type” header is transmitted and a filesystem read stream is piped into the HTTP response stream.

The following examples show files for an exemplary web page:

User Interface: Example HTML file

```
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>File Server usage</title>
  <link rel="stylesheet" href="/index.css">
</head>
<body>
  <h1>This is an example webpage for the file server factory.</h1>
  <p>
    You can view the <a href="/index.css">CSS file</a> and
    the <a href="/index.js">JavaScript file</a> directly.
    <strong>Note:</strong> The client-side JS enforces clicks to be confirmed.
  </p>
  <script src="/index.js"></script>
</body>
</html>
```

User Interface: Example CSS file

```
h1 { font-size: 40px; }
a { color: #000; }
```

User Interface: Example client-side JS file

```
document.addEventListener('click', event => {
  if (event.target.tagName !== 'A') return;
  if (!confirm('really leaving?')) event.preventDefault();
});
```

The web page is served through an HTTP server with an according filesystem interface ([run code](#)):

HTTP: HTTP Filesystem interface factory usage

```
const httpFilesystemInterface = createHttpFilesystemInterface(request => {
  const {pathname} = url.parse(request.url);
  const filePath = pathname === '/' ? '/index.html' : pathname;
  return `${rootDirectory}/http-filesystem-interface-factory${filePath}`;
});
http.createServer(httpFilesystemInterface).listen(50000);
console.log('<iframe src="http://localhost:50000"></iframe>');
```

The example page consists of an HTML file, a stylesheet and client-side JavaScript code. As content, the page contains a headline and a paragraph with hyperlinks to both assets. While the CSS applies basic styling, the JavaScript code intercepts hyperlink clicks and requests confirmation. Following an asset URL results in displaying its file content. The usage code starts with executing the operation `createHttpFilesystemInterface()` and passing in a custom resolver function. This operation prefixes given URL paths with the assets directory. Also, it substitutes the value “/” with “/index.html”. As next step, an HTTP server is instantiated with the returned listener callback and started. Finally, an HTML string is output. Executing the code in the Playground causes to render an iframe with the example page.



Rendering HTML output

The Code Playground passes received information from executed code to the property `innerHTML` of the code output DOM element. Consequently, if a message contains raw HTML, it causes to create actual DOM nodes. This is done on purpose to let code examples render visual elements. Normally, such a behavior should be prevented for security reasons.

Task-based UI

Task-based UI is a User Interface design approach that puts emphasis on a Domain and its use cases. Conceptually, it stands in contrast with the approach **CRUD-based UI**, which focuses on records and data modification. While the pattern is mandatory for software that is based on DDD, it is often applied implicitly. Through the use of a Ubiquitous Language, the User Interface adequately expresses its underlying abstractions. Every meaningful interaction is linked to the execution of a specific use case. When applying CQRS, this means

to create a Command or a Query and to send it to the Application Layer. The messages ensure to capture both intent and data as specific structures. Furthermore, Query results and Domain Events also encapsulate information in a domain-specific format.



Create, read, update, delete

Although the concepts Task-based UI and CRUD-based UI are commonly presented as opposites, they overlap in some aspects. For example, when creating an Entity, a UI often needs to provide controls for all attributes, regardless of design approach. Also, the terms “create”, “read”, “update” and “delete” can and should be used whenever appropriate, even for task-based UIs.

Example: Blog

Consider implementing a blog software that can be used for personal websites. The respective Domain Model defines use cases for writing, publishing, un-publishing and editing individual posts. Writing a new post requires to provide an identity, a title and content. Editing an existing item similarly incorporates an identifier as well as a new title and new content. For publishing a post, the only required argument is the corresponding identity. The action indicates that the content is made available to a public audience. Un-publishing represents the opposite action, which is required for different reasons, such as legal challenges or writing mistakes. This example focuses on the functionalities for writing, publishing and un-publishing posts. For simplicity reasons, it excludes the use case of editing existing items.

The next code implements the Command Handlers component:

Blog software: Command Handlers

```
class BlogPostCommandHandlers {

    #eventStore;

    constructor({eventStore}) {
        this.#eventStore = eventStore;
    }

    handleCommand = createMessageForwarder(this, {messageSuffix: 'Command'});

    async handleWriteBlogPostCommand({data: {blogPostId, title, content}}) {
        await this.#eventStore.save(`blog-post/${blogPostId}`, [{id: generateId(),
            type: 'BlogPostWritten', data: {blogPostId, title, content}}]);
    }
}
```

```
}

async handlePublishBlogPostCommand({data: {blogPostId}}) {
    await this.#eventStore.save(`blog-post/${blogPostId}`,
        [{id: generateId(), type: 'BlogPostPublished', data: {blogPostId}}]);
}

async handleUnpublishBlogPostCommand({data: {blogPostId}}) {
    await this.#eventStore.save(`blog-post/${blogPostId}`,
        [{id: generateId(), type: 'BlogPostUnpublished', data: {blogPostId}}]);
}

}
```

The following example provides a Query Handlers class with lazy Read Model computation:

Blog software: Query Handlers

```
class BlogPostQueryHandlers {

    #eventStore;

    constructor({eventStore}) {
        this.#eventStore = eventStore;
    }

    handleQuery = createMessageForwarder(this, {messageSuffix: 'Query'});

    async handleFindBlogPostsQuery() {
        const {events} = await this.#eventStore.load('$global');
        const blogPostsById = {};
        events.forEach(event => {
            if (event.type === BlogPostWrittenEvent.type) {
                const {blogPostId: id, title, content} = event.data;
                blogPostsById[id] = {id, title, content, isPublished: false};
            }
            if (event.type === BlogPostPublishedEvent.type)
                blogPostsById[event.data.blogPostId].isPublished = true;
            if (event.type === BlogPostUnpublishedEvent.type)
                blogPostsById[event.data.blogPostId].isPublished = false;
        });
        return blogPostsById;
    }
}
```

All relevant state transitions for a blog post are represented by the event types “BlogPostWritten”, “BlogPostPublished” and “BlogPostUnpublished”. For write-related use cases, the class `BlogPostCommandHandlers` implements the according Command Handlers. As constructor argument, it expects an Event Store. Each handler operation transforms given arguments into an event and appends it to a stream. The class `BlogPostQueryHandlers` implements the read-related behavior. For the example, the component simply computes the full Read Model upon each request. Its constructor expects an Event Store instance as argument. The service operation `handleFindBlogPostsQuery()` loads all events from the global stream, creates a blog posts collection and returns it. Effectively, this approach is an on-demand Read Model computation, as explained in Chapter 11. One potential improvement is to introduce a caching behavior.

The next code shows the HTML document for the User Interface:

Blog software: HTML file

```
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Blog</title>
  <link rel="stylesheet" href="/blog/ui/index.css">
</head>
<body>
  <h2>Write blog post</h2>
  <form method="POST" class="write-post">
    <input type="text" name="title" required placeholder="Title" />
    <textarea name="content" cols="30" required placeholder="Content"></textarea>
    <input type="submit" value="Write blog post">
  </form>
  <h2>Written blog posts</h2>
  <section class="blog-posts"></section>
  <script type="module" src="/blog/ui/index.js"></script>
</body>
</html>
```

As preparation for the client-side behavior, the following example implements request helper functions:

Infrastructure code: Browser request helper

```
const getJSON = (url, data) =>
  fetch(`${url}?${createQueryString(data)}`).then(response => response.json());

const post = (url, data) => fetch(url, {method: 'POST',
  headers: {'Content-Type': 'application/json'}, body: JSON.stringify(data)});
```

The HTML file for the blog software consists of two parts. One is the `<form>` element that is used for writing a new blog post. For demonstration purposes, both the `<input>` element and the `<textarea>` element are equipped with a validation attribute. The other part of the HTML document is a container element for rendering all existing blog posts. Also, the markup references a basic stylesheet and the client-side JavaScript as additional assets. The request helper functions `getJSON()` and `post()` are browser-based counterparts to the Node.js variants introduced in Chapter 12. Instead of using the `http` module, the operations utilize the browser Fetch API. In addition to performing a GET request, the operation `getJSON()` also converts a response automatically to JSON.

The next example implements the client-side logic for the UI part:

Blog software: Client-side JavaScript

```
getJSON('/query', {type: 'FindBlogPosts'}).then(postsById => {
  document.querySelector('.blog-posts').innerHTML = Object.values(postsById).map(
    (id, title, content, isPublished) => `<article>
      <strong>${title}</strong> (${isPublished ? 'published' : 'unpublished'})<br>
      <p>${content}</p>
      <button
        data-blog-post-id="${id}"
        data-command-type="${isPublished ? 'Unpublish' : 'Publish'}"
        >${isPublished ? 'Unpublish' : 'Publish'}</button>
    </article>`).join('');
});

document.querySelector('.write-post').addEventListener('submit', async event => {
  const title = event.target.elements.title.value;
  const content = event.target.elements.content.value;
  event.preventDefault();
  await post('/command', {id: generateId(), type: 'WriteBlogPost',
    data: {blogPostId: generateId(), title, content}});
  window.location.reload();
});
```

```
document.querySelector('.blog-posts').addEventListener('click', async event => {
  if (!event.target.hasAttribute('data-command-type')) return;
  const type = `${event.target.getAttribute('data-command-type')}{BlogPost}`;
  const blogPostId = event.target.getAttribute('data-blog-post-id');
  await post('/command', {id: generateId(), type, data: {blogPostId}});
  window.location.reload();
});
```

The code starts with sending a “FindBlogPosts” Query and rendering the received posts. Every rendered `<article>` element consists of title, publishing status, content and a button for toggling the publishing status. Each button receives `data-*` attributes for the respective post identifier and Command type. Next, a submit event listener is registered on the form to write a post. The handler operation retrieves all submitted data and issues a “WriteBlogPost” Command. Also, it prevents form submission by invoking the function `event.preventDefault()`. After the Command execution, the page is reloaded via the operation `window.location.reload()`. Finally, the code registers a click event listener on the post list. For every click on a toggle button, the handler issues either a “PublishBlogPost” or “UnpublishBlogPost” Command and reloads the page.



Differences to a CRUD-based UI

The User Interface implementation for the example provides visual structures that guide a user towards domain-specific use cases. In contrast, a CRUD-based counterpart could provide a single form element, both for writing new posts and editing existing ones. Changing the publishing status of a post would be represented as generic record modification, without adequately capturing the domain-specific meaning.

The next example provides the configuration for the blog software ([run code](#)):

Blog software: Configuration and usage

```
const eventStore = new FilesystemEventStoreWithGlobalStream({storageDirectory});
const commandHandlers = new BlogPostCommandHandlers({eventStore});
const queryHandlers = new BlogPostQueryHandlers({eventStore});

const commandHttpInterface = createHttpInterface(
  message => commandHandlers.handleCommand(message), ['POST']);
const queryHttpInterface = createHttpInterface(
  message => queryHandlers.handleQuery(message), ['GET']);

const rootPage = '/blog/ui/index.html';
const fsInterface = createHttpFilesystemInterface(request => {
  const {pathname} = url.parse(request.url);
  return `${rootDirectory}/..${pathname} === '/' ? rootPage : pathname`;
});

http.createServer((request, response) => {
  const {url} = request;
  if (url.startsWith('/command')) return commandHttpInterface(request, response);
  else if (url.startsWith('/query')) return queryHttpInterface(request, response);
  return fsInterface(request, response);
}).listen(50000);

console.log('<iframe src="http://localhost:50000"></iframe>');


---


```

The code starts with instantiating the components `FilesystemEventStoreWithGlobalStream`, `BlogPostCommandHandlers` and `BlogPostQueryHandlers`. For the Command Handlers and the Query Handlers, the Event Store is used as constructor argument. Afterwards, the HTTP interfaces for Command Handlers and Query Handlers are created using the operation `createHttpInterface()`. Then, an HTTP filesystem interface is constructed via the function `createHttpFilesystemInterface()`. Next, the class `http.Server` is instantiated with a custom listener function. Requests with the path “/command” or “/query” are forwarded to the according Application Layer interface. All other requests are processed by the filesystem interface. Lastly, the server is started and a message is output to render an iframe. Overall, the code exemplifies how to build a User Interface that puts emphasis on a Domain and its use cases.

Optimistic UI

Optimistic UI is a User Interface pattern to improve perceived performance by prematurely indicating the success of state-changing actions. On top of that, the concept is also useful for simulating updates to eventually consistent read data. There are two main scenarios for applying this pattern. The first one is when requesting the execution of a behavior, of which a successful completion should be reported. The second scenario is when a use case execution implies an update to related data that is being displayed. In both cases, a success is indicated regardless of the actual progress. If there is rendered data to update, the originally submitted information can be re-used. However, the User Interface Layer must ensure to not replicate domain-specific data transformations for this purpose.



Labeling tentative information

While the concept of Optimistic User Interfaces clearly differs from a progress visualization, the two patterns can be combined. In addition to an optimistic success indication, there can be a temporary notion that a change has not been confirmed. For example, when sending a chat message, the text may appear as all other messages, only with an additional temporary icon.

Dealing with errors

When building an Optimistic User Interface, an important aspect is how to deal with errors and how to communicate them. While the visible interface parts seemingly ignore the actual progress of an action, it must still be taken into consideration. In case of an error, there are different possibilities. One simple option is to display a generic message and ask the user to reload the interface. This approach has a low complexity and works regardless of the type of action. Another option is to show specialized messages that explain each error individually. On top of that, compensating actions can be offered, such as retrying the original execution. Depending on the criticality of a use case, the User Interface may also require to block further interactions.

Example: Social network

Consider implementing the User Interface for writing posts as part of a social network. The goal is to provide a satisfying user experience through an optimized perceived performance.

Depending on content, client-side bandwidth and server-side load, the process of sending and storing a post can take time. The potential feedback delay can be mitigated by immediately presenting the submitted content as written post. This approach is feasible, as the action has a high success rate. In case of an error, the User Interface must display a message and offer a retry. Upon re-execution, the error message is hidden again. Subsequent failures cause the same behavior to be repeated until the action succeeds. For this example, the server-side part is substituted with a fake implementation.

The first code shows the implementation of a fake Command Handlers component:

Social network: Fake Command Handlers

```
class PostCommandHandlers {  
  
    #successThreshold;  
  
    constructor({chanceOfSuccess}) {  
        this.#successThreshold = chanceOfSuccess / 100;  
    }  
  
    async handleWritePostCommand(command) {  
        return new Promise((resolve, reject) => {  
            setTimeout(() => {  
                if (Math.random() <= this.#successThreshold) resolve();  
                else reject(new Error('server error'));  
            }, Math.random() * 2000);  
        });  
    }  
}
```

The second example provides the HTML document for the User interface:

Social network: HTML document

```
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Business social network</title>
  <link rel="stylesheet" href="/social-network/ui/index.css">
</head>
<body>
  <h2>Write Post</h2>
  <form method="POST" class="write-post">
    <textarea name="content" cols="30" rows="10"></textarea>
    <input type="submit" value="Write post">
  </form>
  <h2>Written Posts</h2>
  <section class="written-posts"></section>
  <script type="module" src="/social-network/ui/index.js"></script>
</body>
</html>
```

The following code implements the client-side behavior:

Social network: Client-side behavior

```
const writtenPostsElement = document.querySelector('.written-posts');

const sendWriteCommand = content =>
  post('/command', {id: generateId(), type: 'WritePost', data: {content}});

document.querySelector('.write-post').addEventListener('submit', async event => {
  event.preventDefault();
  const content = event.target.elements.content.value;
  const postElement = document.createElement('article');
  postElement.innerHTML = content;
  writtenPostsElement.prepend(postElement);
  const response = await sendWriteCommand(content);
  if (response.status === 200) return;
  const errorElement = document.createElement('div');
  errorElement.classList.add('error');
  errorElement.innerHTML = 'Post could not be submitted. <a href="#">Retry?</a>';
  postElement.appendChild(errorElement);
  errorElement.addEventListener('click', async () => {
    errorElement.setAttribute('hidden', 'true');
    const newResponse = await sendWriteCommand(content);
```

```

    if (newResponse.status === 200) errorElement.remove();
    else errorElement.removeAttribute('hidden');
  });
});

```

The class `PostCommandHandlers` provides a fake Command Handlers implementation with a configurable chance of success. Its handler operation `handleWritePostCommand()` utilizes the function `setTimeout()` to emulate a delay. The provided callback randomly either resolves or rejects the overall Promise, depending on the chance of success. As User Interface, the HTML document renders a `<form>` and a post list. The client-side behavior registers a submit event listener. Its callback operation first uses the submitted content to render a post. Only afterwards, it issues the command “WritePost”. If the request succeeds, the function exits. In case of an error, a message and a re-execution link are displayed. Also, a click event listener is registered to issue another command. If the retried request fails, the error is displayed again.

The final code provides the configuration for the write post functionality ([run code](#)):

Social network: Configuration and usage

```

const commandHandlers = new PostCommandHandlers({chanceOfSuccess: 25});

const commandHttpInterface = createHttpInterface(
  message => commandHandlers.handleWritePostCommand(message), ['POST']);

const fsInterface = createHttpFilesystemInterface(request => {
  const {pathname} = url.parse(request.url);
  const indexPath = '/social-network/ui/index.html';
  return `${rootDirectory}/..${pathname === '/' ? indexPath : pathname}`;
});

http.createServer((request, response) => {
  if (request.method === 'POST') return commandHttpInterface(request, response);
  return fsInterface(request, response);
}).listen(50000);

console.log('<iframe src="http://localhost:50000"></iframe>');

```

The configuration code starts with instantiating the class `PostCommandHandlers` with a success chance of 25. Note that with an actual implementation, this chance must be much higher for an Optimistic UI to make sense. As next step, an HTTP interface is created by executing the operation `createHttpInterface()`. Then, an HTTP filesystem interface is instantiated via the

function `createHttpFilesystemInterface()`. Afterwards, a server is created and started. The provided listener operation forwards all POST requests to the Command Handler interface and all others to the filesystem. Finally, a console message is logged to render an `<iframe>` element. When submitting the write post form, the feedback is given instantaneously, regardless of the actual progress. Only when a request fails, the error message and the re-execution link are displayed.

Reactive Read Models

Instead of a static data structure, Read Models can also be represented as a continuous source of information. This approach is especially useful for building reactive User Interfaces. Typically, a read-related use case is treated as a one-time operation that returns a single data set. Consequently, each consumer must actively check for potential updates at given points. As an alternative to plain data, a mechanism can be provided that incorporates both current information and future updates. This approach works best in combination with an event-based implementation, such as when applying Event Sourcing. Also, Reactive Read Models can help to enable real-time collaboration. However, due to their additional complexity, they should only be used when there is an actual need for it.

Streams

Reactive Read Models require a mechanism to notify consumers of future updates. There are different patterns for this purpose, such as callbacks, asynchronous generators and streams. The Node.js module `stream` provides components to work with streaming data. Compared to plain callbacks and generator functions, it offers advanced functionalities, such as data throttling and chaining. Also, its use enables easier integration with other native Node.js parts. The module `stream` exposes four types of streams. The type `Readable` represents a consumable source of data. Destinations to send information to are implemented with the class `Writable`. The type `Duplex` is both readable and writable, which is useful for bidirectional communication. Finally, consuming information from one source, transforming it and forwarding the result is achieved with the class `Transform`.

The following example shows an exemplary usage of the stream classes `Readable` and `Transform` ([run code](#)):

Stream: Usage

```
class ObjectToStringStream extends Transform {

  constructor() {
    super({objectMode: true});
  }

  _transform(input, _, callback) {
    callback(null, `${JSON.stringify(input)}\n`);
  }

}

const readableStream = new Readable({objectMode: true, read() {}});
const pushNextEvent = () => {
  readableStream.push({type: 'MessageReceived', id: generateId()});
  setTimeout(pushNextEvent, Math.random() * 3000);
};
pushNextEvent();

readableStream.pipe(new ObjectToStringStream()).pipe(process.stdout);
```

First, the code defines the class `ObjectToStringStream`. The component extends the type `Transform` and invokes its base constructor with the `objectMode` option set to `true`. This enables to also process objects instead of only strings and buffers. Every transform stream implements its behavior in the operation `_transform()`, which in this case converts objects to JSON strings. The component definition is followed by instantiating the class `Readable` with an activated `objectMode` option and an empty `read()` function. Afterwards, the recursive operation `pushNextEvent()` is defined and executed, which continuously pushes messages into the readable stream. Finally, the stream is connected to an `ObjectToStringStream` instance, which itself is linked to the standard output. This is done via the function `pipe()`. Executing the code causes to output messages repeatedly.



Data buffering

Every Node.js stream is equipped with an internal in-memory buffer mechanism. This allows readable streams to read underlying data independent of consumption and writable streams to accept input independent of processing. While the buffers have advisory size thresholds, there is no hard limit enforced. For the sake of simplicity, the custom stream implementations in this chapter ignore buffer thresholds.

Server-sent Events

Server-sent Events (SSE) is a unidirectional communication protocol on top of HTTP for receiving messages over a persistent connection. One approach to mimic this behavior is Long Polling, where individual subsequent connections are established for each message to receive. The advantage of SSE is that it maintains a single long-lasting connection with an automatic reconnect mechanism. The standard defines the client-side interface `EventSource`, which expects a URL and enables to register named event listeners. Server-sent HTTP responses must be equipped with the “Content-Type” header value “text/event-stream”. Every message to send consists of an optional identifier, an optional type with the default value “message” and custom data. Each field is represented as an individual line with key and value, whereas messages are separated by an empty line.

Server-sent Events example

```
data: {userId: 'user-1', email: 'foo@example.com'}
```

```
type: UserLoggedIn
data: {userId: 'user-1', device: 'laptop'}
```

```
id: 1234
type: UserLoggedOut
data: {userId: 'user-1'}
```



Idempotent processing

When consuming Server-sent Events, the processing mechanism should ideally be idempotent. This is because whenever an HTTP connection is interrupted and re-established, previously processed messages may appear again. Alternatively, event duplication can be mitigated through the HTTP header “Last-Event-ID”, which is automatically sent from the `EventSource` component.

The following code implements a transform stream for the Server-sent Events format:

HTTP: SSE Transform Stream

```
class ServerSentEventStream extends Transform {  
  
    constructor() {  
        super({objectMode: true});  
    }  
  
    _transform(data, _, callback) {  
        callback(null, `data: ${JSON.stringify(data)}\n\n`);  
    }  
  
}
```

The next example shows an extended version of the HTTP interface factory with SSE support:

HTTP: HTTP interface factory with SSE support

```
const createHttpInterfaceWithSseSupport = (targetFunction, methods = []) =>  
    async (request, response) => {  
        try {  
            if (!methods.includes(request.method)) throw new Error('invalid method');  
            const message = request.method === 'POST' ?  
                JSON.parse(await parseHttpStream(request)) : parseQueryString(request.url);  
            const result = await targetFunction(message);  
            const resultIsStream = result && result.readable && result._read;  
            response.writeHead(200, {'Cache-Control': 'no-cache',  
                'Content-Type': resultIsStream ? 'text/event-stream' : 'application/json'});  
            if (resultIsStream) {  
                response.connection.setTimeout(0);  
                result.pipe(new ServerSentEventStream()).pipe(response);  
            } else response.end(result != null ? JSON.stringify(result) : result);  
        } catch (error) {  
            response.writeHead(500);  
            response.end(error.toString());  
        }  
    };
```

The component `ServerSentEventStream` enables to serialize a stream of objects into the Server-sent Events format. The class extends the type `Transform` and invokes the base constructor with an enabled `objectMode` flag. Its operation `_transform()` converts an object into a

data-only event. Dedicated event identifiers are not required and the absence of custom types allows a single listener for the default type. The reworked factory `createHttpInterface()` extends the original with SSE support. After each target function invocation, the type of the return value is determined. In case of a stream, the header “Content-Type” is adjusted, the connection timeout is increased and SSE messages are transmitted. This is done by piping the returned stream into a `ServerSentEventStream` instance and connecting the result with the HTTP response.

The third example illustrates the usage of the previously introduced components ([run code](#)):

HTTP: Usage of HTTP interface factory with SSE support

```
const handleStreamRandomNumberQuery = async () => {
  const stream = new Readable({objectMode: true, read() {}});
  setInterval(() =>
    stream.push({id: generateId(), type: 'RandomNumber', data: Math.random()}),
    1000);
  return stream;
};

const httpInterface =
  createHttpInterfaceWithSseSupport(handleStreamRandomNumberQuery, ['GET']);

http.createServer((request, response) => {
  if (request.url.startsWith('/stream')) return httpInterface(request, response);
  response.writeHead(200, {'Content-Type': 'text/html'});
  response.end(`<html>
    <head><title>SSE example</title></head>
    <body>
      <pre></pre>
      <script>
        const eventSource = new EventSource('/stream');
        eventSource.addEventListener('message', message =>
          document.querySelector('pre').innerHTML += message.data + '\\n');
      </script>
    </body>
  </html>`);
}).listen(50000);

console.log('<iframe src="http://localhost:50000"></iframe>');
```

The code starts with defining the Query Handler function `handleStreamRandomNumberQuery()`. This operation returns a stream that continuously transmits “RandomNumber” events. As

next step, an HTTP interface with SSE support is created via the reworked factory function. Then, an HTTP server is instantiated. The passed in listener operation forwards requests for URLs starting with “/stream” to the Query Handler interface. For all other requests, an HTML document is returned as response. The therein contained JavaScript code first creates an EventSource instance with the relative URL “/stream”. Then, it registers an event listener for the default type “message”. The according callback operation appends each received message as plain text to a `<pre>` element. Executing the code in the Playground shows a page that renders SSE messages.

Architectural approaches

Independent of infrastructural patterns and technologies, there are different architectural approaches for operating Reactive Read Models. The following subsections explain and illustrate three possibilities, each with their own advantages and disadvantages. All of them share the fact that the Application Layer responds to a selected Query type with a stream result. The difference lies in what information the respective streams transmit. While there are more approaches, the selected ones represent the most common and distinct choices. The subsections are preceded by the introduction of an example topic, which is used to illustrate each approach individually. As side effect, the example implementations provide real-time collaboration capabilities. For this aspect, the goal is to demonstrate that Reactive Read Models enable this functionality automatically to some extent.

Example: Todo list

Consider implementing a todo list software. The corresponding Domain Model consists of two parts. One is the todo list component, which encloses an identifier and a mutable collection of todos. The second part is the todo itself, which contains an identity, content and a completion status. Both the content and completion status can be changed. The todo list component represents the root Entity of a transactional boundary that encloses itself and all associated todos. The relevant use cases consist of writing a todo, marking a todo as completed and undoing a completion. Furthermore, the User Interface should enable a collaborative use of individual lists. This means, whenever a use case is executed, the resulting state change must be propagated to all affected clients.

The first example provides the necessary event definitions:

Todo list: Events

```
const TodoWrittenEvent = createEventType('TodoWritten',
  {todoListId: 'string', todoId: 'string', content: 'string'});

const TodoCompletedEvent = createEventType('TodoCompleted',
  {todoListId: 'string', todoId: 'string'});

const TodoUncompletedEvent = createEventType('TodoUncompleted',
  {todoListId: 'string', todoId: 'string'})
```

The second example implements the required Command Handlers component:

Todo list: Command Handlers

```
class TodoListCommandHandlers {

  #eventStore;

  constructor({eventStore}) {
    this.#eventStore = eventStore;
  }

  handleCommand = createMessageForwarder(this, {messageSuffix: 'Command'});

  async handleWriteTodoCommand({data: {todoListId, todoId, content}}) {
    await this.#eventStore.save(`todo-list/${todoListId}`,
      [new TodoWrittenEvent({todoListId, todoId, content})]);
  }

  async handleCompleteTodoCommand({data: {todoListId, todoId}}) {
    await this.#eventStore.save(`todo-list/${todoListId}`,
      [new TodoCompletedEvent({todoListId, todoId})]);
  }

  async handleUncompleteTodoCommand({data: {todoListId, todoId}}) {
    await this.#eventStore.save(`todo-list/${todoListId}`,
      [new TodoUncompletedEvent({todoListId, todoId})]);
  }
}
```

The event definitions express the relevant state changes for the todo list software. In addition to a list identity, each of the types also incorporates a todo identifier. On top of that, the

“TodoWritten” event type includes the submitted content. Note that there is no dedicated message for the creation of a list. This is because an instance counts as existing as soon as its first event occurs. The class `TodoListCommandHandlers` implements all the required Command Handler operations. Its constructor expects an Event Store instance and invokes the function `createMessageForwarder()` for exposing a unified message handler. Similar to the blog example, each use case operation exclusively creates an event and appends it to the affected stream.

The next code provides a basic HTML document:

Todo list: HTML document

```
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Todo List</title>
</head>
<body>
  <h1>Todo List</h1>
  <h2>Add new todo</h2>
  <form method="POST" class="write-todo">
    <input type="text" name="content">
    <input type="submit" value="Write todo">
  </form>
  <h2>Todos</h2>
  <section class="todos"></section>
  <script type="module" src="/todo-list/shared/ui/index.js"></script>
</body>
</html>
```

This is followed by the client-side JavaScript code:

Todo list: Client-side setup

```
const parameters = new URLSearchParams(location.search);

const todoListId = parameters.get('todo-list-id');
const approach = parameters.get('approach');

document.querySelector('.write-todo').addEventListener('submit', async event => {
  const content = event.target.elements.content.value;
  post('/command', {id: generateId(), type: 'WriteTodo', data:
    {todoListId, todoId: generateId(), content}});
  event.preventDefault();
```

```
});

document.querySelector('.todos').addEventListener('click', event => {
  if (!event.target.matches('input[type=checkbox]')) return;
  const todoId = event.target.closest('[id]').getAttribute('id');
  const type = event.target.checked ? 'CompleteTodo' : 'UncompleteTodo';
  post('/command', {id: generateId(), type, data: {todoListId, todoId}});
});

import(`../../${approach}/ui/index.js`);
```

The HTML document contains a headline, an input form, a todo list container element and a `<script>` element. The client-side code starts with retrieving the values of the URL parameters “todo-list-id” and “approach”. This is done with the native browser interface `URLSearchParams`. Next, a form submit event listener is registered. Its callback operation issues a “WritePost” Command with the respectively entered content. Afterwards, a click event listener is registered on the list container. Its handler operation first verifies that the event occurred on a checkbox. Then, it retrieves the todo identity from an ancestor node. As next step, it either issues a “CompleteTodo” or a “UncompleteTodo” Command, depending on the checkbox status. Finally, the code dynamically imports the module for the specific implementation approach.

The next example provides a helper function to render a todo list:

Todo list: Todo rendering

```
const renderTodos = todos => {
  const todosElement = document.querySelector('.todos');
  Object.values(todos).forEach(({id, content, isCompleted}) => {
    const todoElement = todosElement.querySelector(`[id="${id}"]`);
    if (!todoElement) {
      const newTodoElement = document.createElement('article');
      newTodoElement.setAttribute('id', id);
      newTodoElement.innerHTML = `<label>
        <input type="checkbox" ${isCompleted ? 'checked' : ''}>
        <span>${content}</span>
      </label>`;
      todosElement.appendChild(newTodoElement);
    } else {
      const contentElement = todoElement.querySelector('span');
      if (contentElement.innerHTML !== content) contentElement.innerHTML = content;
      const inputElement = todoElement.querySelector('input');
```

```

    if (inputElement.checked !== isCompleted) inputElement.checked = isCompleted;
}
});
};

}

```

The last code shows a helper operation to create a server for the todo list software:

Todo list: Server factory

```

const createServer = ({commandHandlers, queryHandlers}) => {
  const commandHttpInterface = createHttpInterface(
    message => commandHandlers.handleCommand(message), ['POST']);
  const queryHttpInterface = createHttpInterfaceWithSseSupport(
    message => queryHandlers.handleQuery(message), ['GET']);

  const fsInterface = createHttpFilesystemInterface(request => {
    const {pathname} = url.parse(request.url);
    const indexPath = '/todo-list/shared/ui/index.html';
    return `${rootDirectory}/../../${pathname === '/' ? indexPath : pathname}`;
  });

  return http.createServer((request, response) => {
    const {url} = request;
    if (url.startsWith('/command')) return commandHttpInterface(request, response);
    if (url.startsWith('/query')) return queryHttpInterface(request, response);
    return fsInterface(request, response);
  });
};

```

The operation `renderTodos()` renders a group of todo objects as HTML. For each entry, it first checks whether an according element exists. If not, a new `<article>` node is added with an “id” attribute and both a checkbox and content as children. If an element already exists, the checkbox and the content are conditionally updated. The function `createServer()` sets up an HTTP server. First, it instantiates interfaces for the given Command Handlers and Query Handlers. For the Query part, it uses the operation `createHttpInterfaceWithSseSupport()`. Also, a filesystem interface is created. Finally, the function `http.createServer()` is executed. While requests with the path “/command” are sent to the Command Handlers, the path “/query” triggers a Query Handler execution. All other requests are forwarded to the filesystem interface.

Event forwarding

One option for a Reactive Read Model is to consume an event stream directly in the User Interface client part. For that purpose, a Query Handler forwards all information from an originating event source. With this approach, the client can independently operate a projection and freely maintain a custom data structure. One disadvantage is that domain-specific events are exposed to the outside. Although User Interface parts architecturally belong with the context of their associated use cases, this circumstance may be undesirable. Also, there can be a network traffic overhead when the original event source contains extraneous data. Furthermore, the approach only works for streams that exclusively enclose information a respective user is allowed to access. Often, this is not the case for global event streams.



Persistent client projections

The client part of a User Interface can operate persistent projections in order to reduce network traffic or computational overhead. For example, a browser can save a Read Model in its local storage. In case of Event Sourcing, the Query for exposing an event stream can incorporate a stream start version argument. This enables to exclude already processed events.

The first code provides a Query Handlers component to expose a todo list event stream:

Event forwarding: Query Handlers

```
class TodoListQueryHandlers {

    #eventStore;

    constructor({eventStore}) {
        this.#eventStore = eventStore;
    }

    handleQuery = createMessageForwarder(this, {messageSuffix: 'Query'});

    async handleStreamTodoListEventsQuery({data: {todoListId}}) {
        const todoListEventStream = new Readable({objectMode: true, read() {}});
        this.#eventStore.subscribe(
            `todo-list/${todoListId}`, event => todoListEventStream.push(event));
        return todoListEventStream;
    }
}
```

```
}
```

The next example shows the client-side behavior for consuming an event stream:

Event forwarding: Client-side behavior

```
const parameters = new URLSearchParams(location.search);
const todoListId = parameters.get('todo-list-id');

const todos = {};

const queryString = createQueryString(
  {type: 'StreamTodoListEvents', data: {todoListId, startVersion: 1}});
const eventSource = new EventSource(`/query?${queryString}`);

eventSource.addEventListener('message', ({data: serializedEvent}) => {
  const {type: eventType, data} = JSON.parse(serializedEvent);
  if (eventType === 'TodoWritten')
    todos[data.todoId] = {id: data.todoId, content: data.content};
  if (eventType === 'TodoCompleted') todos[data.todoId].isCompleted = true;
  if (eventType === 'TodoUncompleted') todos[data.todoId].isCompleted = false;
  renderTodos(todos);
});
```

The class `TodoListQueryHandlers` enables to access an individual todo list event stream. Its operation `handleStreamTodoListEventsQuery()` creates a readable stream, registers an event store subscriber to forward all events, and returns the stream. The client-side behavior starts with retrieving the todo list identifier from the URL query string. Also, it defines the object `todos` for containing the Read Model data. Then, it constructs and serializes a “`StreamTodoListEvents`” Query, combines the result with a URL path and instantiates the `EventSource` class. Next, a message event listener is registered. Its callback operation first de-serializes the forwarded event from the received message data. Then, it executes the projection logic to update the Read Model. Finally, it invokes the function `renderTodos()` with the `todos` variable to update the User Interface.

The third example provides the setup code of the previously introduced components ([run code](#)):

Event forwarding: Server configuration

```
const eventStore = new FilesystemEventStoreWithGlobalStream({storageDirectory});

const commandHandlers = new TodoListCommandHandlers({eventStore});
const queryHandlers = new TodoListQueryHandlers({eventStore});

createServer({commandHandlers, queryHandlers}).listen(50000);

const url = 'http://localhost:50000/?approach=event-forwarding&todo-list-id=123';
console.log(`<iframe src="${url}"></iframe>`);
```

The setup code starts with creating an Event Store instance. This is followed by instantiating both the shared Command Handlers component and the custom Query Handlers class. Next, a server is created using the operation `createServer()` with the Application Services as arguments. Then, a URL is defined that includes both the approach name and an exemplary todo list identity. Finally, a message with an `<iframe>` element is logged to the console. Executing the code in the Playground provides a fully functional todo list software that can be used collaboratively. Independent of which user issues a Command execution, all clients receive a message and update their User Interface. Inspecting the network traffic shows that every received message contains the full event from the Event Store.

Data events

Another option for a Reactive Read Model is to provide streams that continuously transmit the complete derived data structure. In this case, the Application Layer operates a projection that consumes the event source and maintains a Read Model. In response to an according Query type, a dedicated stream is returned and the current derived data is initially transmitted. Upon each relevant event, the Read Model is updated and the data structure is pushed to all active Query streams. This approach reduces the complexity for the User Interface, as it only consumes derived data. One downside is that if the User Interface requires Read Model adjustments, it also affects other architectural parts. Generally, the approach makes most sense for moderately sized data structures that change infrequently.

The first code shows a dedicated Read Model projection component:

Data events: Read Model projection

```
class TodoListReadModelProjection {  
  
    #eventStore; #todoListReadModelStorage; #todoListReadModelMessageBus;  
  
    constructor({eventStore, todoListReadModelStorage, todoListReadModelMessageBus}) {  
        this.#eventStore = eventStore;  
        this.#todoListReadModelStorage = todoListReadModelStorage;  
        this.#todoListReadModelMessageBus = todoListReadModelMessageBus;  
    }  
  
    activate() {  
        this.#eventStore.subscribe('$global', ({type, data}) => {  
            if (!this.#todoListReadModelStorage.has(data.todoListId))  
                this.#todoListReadModelStorage.set(data.todoListId, {});  
            const todoList = this.#todoListReadModelStorage.get(data.todoListId);  
            if (type === 'TodoWritten')  
                todoList[data.todoId] = {id: data.todoId, content: data.content};  
            if (type === 'TodoCompleted') todoList[data.todoId].isCompleted = true;  
            if (type === 'TodoUncompleted') todoList[data.todoId].isCompleted = false;  
            this.#todoListReadModelMessageBus.publish(`TodoList/${data.todoListId}`,  
                this.#todoListReadModelStorage.get(data.todoListId));  
        });  
    }  
}  


---


```

The next example provides the according Query Handlers class:

Data events: Query Handlers

```
class TodoListQueryHandlers {  
  
    #todoListReadModelStorage; #todoListReadModelMessageBus;  
  
    constructor({todoListReadModelStorage, todoListReadModelMessageBus}) {  
        this.#todoListReadModelStorage = todoListReadModelStorage;  
        this.#todoListReadModelMessageBus = todoListReadModelMessageBus;  
    }  
  
    handleQuery = createMessageForwarder(this, {messageSuffix: 'Query'});  
  
    async handleStreamTodoListStateQuery({data: {todoListId}}) {
```

```
const todoListEventStream = new Readable({objectMode: true, read() {}});
todoListEventStream.push(this.#todoListReadModelStorage.get(todoListId) || {});
this.#todoListReadModelMessageBus.subscribe(`TodoList/${todoListId}`,
  readModel => todoListEventStream.push(readModel));
return todoListEventStream;
}

}
```

The class `TodoListReadModelProjection` is responsible for maintaining todo list Read Models and publishing update notifications via a dedicated Message Bus. As constructor arguments, it expects an Event Store, a storage component and a Message Bus. Its operation `activate()` subscribes a listener to the global event stream. The passed in callback updates the Read Model for a given identity and publishes an update notification to the Message Bus. The component `TodoListQueryHandlers` provides the necessary Query Handler operation. Its constructor expects a Read Model storage and a Message Bus. The function `handleStreamTodoListStateQuery()` instantiates a stream, transmits the initial Read Model, subscribes to the Message Bus and returns the stream. Upon each Read Model update notification, it forwards the latest data to the stream.

The next example shows the client-side behavior for consuming the data event stream:

Data events: Client-side behavior

```
const todoListId = parameters.get('todo-list-id');

const queryString = createQueryString(
  {type: 'StreamTodoListState', data: {todoListId}});
const eventSource = new EventSource(`/query?${queryString}`);
eventSource.addEventListener('message',
  ({data: serializedData}) => renderTodos(JSON.parse(serializedData)));
```

The last code provides the server-side setup for the second approach ([run code](#)):

Data events: Server setup

```
const eventStore = new FilesystemEventStoreWithGlobalStream({storageDirectory});

const todoListReadModelStorage = new Map();
const todoListReadModelMessageBus = new MessageBus();

const commandHandlers = new TodoListCommandHandlers({eventStore});
const queryHandlers = new TodoListQueryHandlers(
  {todoListReadModelStorage, todoListReadModelMessageBus});
const readModelProjection = new TodoListReadModelProjection(
  {eventStore, todoListReadModelStorage, todoListReadModelMessageBus});
readModelProjection.activate();

createServer({commandHandlers, queryHandlers}).listen(50000);

const url = 'http://localhost:50000/?approach=data-events&todo-list-id=123';
console.log(`<iframe src="${url}"></iframe>`);
```

The client-side code again starts with retrieving the URL query parameter for the todo list identity. Next, it constructs and serializes a “StreamTodoListState” message. Then, an `EventSource` instance is created using a URL path together with the serialized `Query`. Afterwards, a message event listener is added that de-serializes the transmitted event and invokes the function `renderTodos()`. The server-side setup again instantiates an Event Store, the shared Command Handlers component and the specialized Query Handlers class. On top of that, it also creates instances for a Read Model storage, a Message Bus and the Read Model projection. Compared to the first approach, the client-side code is significantly less complex. At the same time, the network traffic is increased due to always sending the full Read Model.

Change events

The third approach for Read Model reactivity uses an initial event with the full data structure and subsequent change messages. As with the second approach, the Application Layer is responsible for maintaining the Read Model. Also, the according Query Handler responds with a dedicated readable stream that initially transmits the current data. However, instead of pushing the full Read Model continuously, each change triggers a message that exclusively describes the updated parts. This results in an optimal network traffic usage. Also, no internal domain-specific events are exposed to the outside. Compared to the previous approach, the complexity in the User Interface part is slightly increased. This is because it must contain the logic for translating change descriptions into update operations of the Read Model data.



Standardized change descriptions

The structure and content of messages that represent a Read Model change can be chosen freely. At the same time, there are various standards for describing data changes. One example is the IETF RFC-6902 that specifies the JSON Patch format. Also, many storage technologies already provide notifications with change descriptions that can simply be re-used.

The first example shows the Read Model projection component:

Change events: Read Model projection

```
class TodoListReadModelProjection {  
  
    #eventStore; #todoListReadModelStorage; #todoListReadModelMessageBus;  
  
    constructor({eventStore, todoListReadModelStorage, todoListReadModelMessageBus}) {  
        this.#eventStore = eventStore;  
        this.#todoListReadModelStorage = todoListReadModelStorage;  
        this.#todoListReadModelMessageBus = todoListReadModelMessageBus;  
    }  
  
    activate() {  
        this.#eventStore.subscribe('$global', ({type, data}) => {  
            if (!this.#todoListReadModelStorage.has(data.todoListId))  
                this.#todoListReadModelStorage.set(data.todoListId, {});  
            const todoList = this.#todoListReadModelStorage.get(data.todoListId);  
            if (type === 'TodoWritten')  
                todoList[data.todoId] = {id: data.todoId, content: data.content};  
            if (type === 'TodoCompleted') todoList[data.todoId].isCompleted = true;  
            if (type === 'TodoUncompleted') todoList[data.todoId].isCompleted = false;  
            this.#todoListReadModelMessageBus.publish(  
                `TodoList/${data.todoListId}`, {[data.todoId]: todoList[data.todoId]});  
        });  
    }  
}
```

The second example outlines the associated Query Handlers class:

Change events: Query Handlers

```
class TodoListQueryHandlers {  
  
    constructor({todoListReadModelStorage, todoListReadModelMessageBus}) { /* .. */ }  
  
    async handleStreamTodoListChangesQuery({data: {todoListId}}) { /* .. */ }  
  
}
```

The reworked version of the component `TodoListReadModelProjection` maintains todo list Read Models and sends generic change notifications. Similar to the projection of the previous implementation, it publishes a message to the dedicated Message Bus after each data modification. However, instead of sending the full Read Model, it provides a change object that exclusively contains the modified todo. Note that this object has the same structure as a full todo list Read Model. The goal is to simplify the processing logic of the associated stream consumers. Using an identical structure for the full data and subsequent changes allows to treat the messages in the same way. The class `TodoListQueryHandlers` provides the Query Handler operation `handleStreamTodoListChangesQuery()`. Its implementation is identical to the previous approach.

The next example shows the client-side behavior for processing the data and the change events ([run code usage](#)):

Change events: Client-side behavior

```
const parameters = new URLSearchParams(location.search);  
const todoListId = parameters.get('todo-list-id');  
  
const todos = {};  
const queryString = createQueryString(  
    {type: 'StreamTodoListChanges', data: {todoListId}});  
  
const eventSource = new EventSource(`/query?${queryString}`);  
eventSource.addEventListener('message', ({data: serializedMessage}) => {  
    const todosToUpdate = JSON.parse(serializedMessage);  
    Object.assign(todos, todosToUpdate);  
    renderTodoList(todos);  
});
```

As with the other approaches, the client-side behavior first retrieves the todo list identifier from the URL. Next, the variable `todos` is defined for the Read Model data. Then, a

Query is created and serialized, which is used for instantiating the `EventSource` class. Afterwards, a message event listener is registered. The callback operation first de-serializes the actual message. Then, it assigns all contained properties to the Read Model using the operation `Object.assign()`. This works for both the initial full data set and subsequent change descriptions, as their structure is identical. Finally, the function `renderTodos()` is executed with the current Read Model. The server-side setup code is identical to the previous implementation. Overall, this approach represents a combination of moderate client-side complexity and optimized network traffic.



Omitting the intermediary data

Maintaining a separate data object, which is passed to a generic render function, can result in a cleanly separated architecture. At the same time, it can increase code complexity unnecessarily and also degrade render performance. As alternative, the handler operations of an `EventSource` instance that receive change events can also directly update specific UI elements.

The three explained and illustrated implementation approaches for Reactive Read Models all have their individual benefits and implications. Even more, each of them can prove to be superior for certain use cases. As a consequence, none of them can be recommended in general. Also, as explained previously, there are even further possibilities to implement Reactive Read Models.

Components and composition

Every User Interface is a composition of components with individual responsibilities and clear boundaries. The underlying foundation is always a component model that defines structural and behavioral possibilities. Such a model is a combination of conceptual and technical aspects. While the technical approach implies the capabilities, the conceptual model establishes the necessary context. Generally, there are two categories of components. For one, there are parts that are concerned with generic presentational aspects. Secondly, there are parts that additionally host domain-specific functionalities. Regardless of categories, each element encapsulates its content, presentation and behavior. Also, components themselves can be a composition of others. Architecturally speaking, domain-specific parts always belong to their respective context. Whenever components of different contexts are composed with each other, they form a relationship.

Custom Elements

Custom Elements is a web standard that enables to build specialized HTML elements on top of existing native ones. Essentially, it exposes the internal browser component model. One key advantage of the standard is that custom parts integrate seamlessly with native ones due to the same technology. Creating a Custom Element requires to implement a class that extends an existing HTML element type. In most cases, this should be the generic constructor `HTMLElement`. More specific ones, such as `HTMLOptionElement`, are necessary when building drop-in replacements for native elements. Every custom implementation must be registered in the Custom Elements registry via the function `customElements.define()`. The operation expects a tag name, a constructor and optional options. The options are required when extending a specific HTML element.



Relation to Web Components

Web Components is an umbrella term that encloses multiple web standards for building specialized HTML elements. This includes Custom Elements, Shadow DOM and HTML Templates. However, many times the term is used as synonym for Custom Elements. One reason is that this particular standard enables most of the required functionality and the others are optional.

Every Custom Element has a lifecycle and associated events, for which callback functions can be defined. Whenever an element is appended to a node that is connected to a document, the operation `connectedCallback()` is executed. Instead of the constructor, all initialization work, such as fetching resources or rendering children, should be done inside this callback. Since the callback can execute multiple times, one-time setup must be protected from repeated execution. The operation `disconnectedCallback()` is invoked when an element is removed from a document-connected node. Custom Elements can configure change notifications for attributes by defining the static getter function `get observedAttributes()`. Whenever one of the attributes change, the function `attributeChangedCallback()` is invoked. Finally, the operation `adoptedCallback()` is executed when an element is moved to another document.

The following code provides an exemplary Custom Element implementation for a simple calculator:

Calculator: Custom Element

```
class CalculatorElement extends HTMLElement {  
  
    static get observedAttributes() { return ['a', 'operator', 'b']; }  
  
    attributeChangedCallback() {  
        const attributes = CalculatorElement.observedAttributes;  
        const [a, operator, b] = attributes.map(name => this.getAttribute(name));  
        this.innerHTML = new Function(`return ${a} ${operator} ${b}`)();  
    }  
  
}  
  
customElements.define('simple-calculator', CalculatorElement);
```

The next example shows an HTML document for an exemplary usage of the calculator element ([run code usage](#))

Calculator: HTML usage

```
<html lang="en">  
  <head>  
    <meta charset="UTF-8">  
    <title>Simple calculator</title>  
  </head>  
  <body>  
    <simple-calculator a="3" operator="*" b="9"></simple-calculator>  
    <script src=".index.js"></script>  
  </body>  
</html>
```

The class `CalculatorElement` implements a Custom Element for a simple calculator and extends the generic class `HTMLElement`. As observed attributes, it specifies the names “a”, “operator” and “b”. Whenever one of the attributes change, the function `attributeChangedCallback()` is executed. This also includes initial value assignments. The lifecycle callback operation starts with retrieving all attribute values. Then, it creates and evaluates an according mathematical expression using the `Function()` constructor. Next, the computation result is assigned to the `innerHTML` property of the element. For registering the Custom Element, the function `customElements.define()` is invoked with the tag name “simple-calculator”. The HTML file illustrates an exemplary tag usage. Viewing the document shows that the

calculator renders the correct result. Furthermore, when changing its attributes dynamically, the expression is re-evaluated.



Valid names and non-void tags

There are two important rules to follow for the name and the usage of a Custom Element tag. The name that is passed to `customElements.define()` must contain at least one hyphen character. This is done to prevent collisions with native element names. The second rule is that a Custom Element may not be written as self-closing tag.

Example: Online newspaper

Consider implementing the User Interface for the home page of an online newspaper. The goal is to display an overview of all latest articles. For the example, the relevant part of the Domain Model consists of two components. One is the article, which is a combination of title, author identity, summary and content. The second Domain Model component is the author, which consists of a name and personal information. For the home page, each article is rendered as a preview item with title, author and summary text. The author part includes the name and a tooltip with the personal information. This means that the data for a single article preview originates from two separate Entities. For brevity reasons, the example uses plain objects as data source.

The first code shows the data for authors and articles:

Online newspaper: Data

```
const authors = [
  {id: '1', name: 'Jack Johnson', info: 'Passionate cook'},
  {id: '2', name: 'Jill Jackson', info: 'Loves to surf'},
];

const articles = [
  {id: '3', title: 'Changing weather', authorId: '1',
   summary: 'How the weather changed over the years', content: '...'},
  {id: '4', title: 'Shiny new globalization', authorId: '1',
   summary: 'The pros and cons of a globalized economy', content: '...'},
  {id: '5', title: 'How sustainable are e-books', authorId: '2',
   summary: 'Are e-books better for the environment?', content: '...'},
];
```

The next example provides a Query Handlers component for retrieving all required information:

Online newspaper: Query Handlers

```
class NewspaperQueryHandlers {  
  
    handleQuery = createMessageForwarder(this, {messageSuffix: 'Query'});  
  
    async handleFindLatestArticlesQuery() {  
        return articles.map(article => article.id);  
    }  
  
    async handleGetArticleQuery(message) {  
        return articles.find(summary => summary.id === message.data.articleId);  
    }  
  
    async handleGetAuthorQuery(message) {  
        return authors.find(info => info.id === message.data.authorId);  
    }  
}
```

The source code for the data defines the two arrays `authors` and `articles`. For the authors list, there are two individual entries defined. The `articles` array contains three separate items, of which two reference the same author identity. These data structures can be understood as Read Model information, as they are exclusively used for read-related functionalities. The class `NewspaperQueryHandlers` implements the required Query Handlers for the homepage use cases. Its constructor creates a unified Query Handler interface via the function `createMessageForwarder()`. Retrieving all latest article identifiers is done by executing the operation `handleFindLatestArticlesQuery()`. The function `handleGetArticleQuery()` returns the article entry for a given identifier. Finally, author data can be retrieved with the operation `handleGetAuthorQuery()`.

The following code shows the implementation for a tooltip Custom Element:

Online newspaper: Info Tooltip

```
class InfoToolTipElement extends HTMLElement {  
  
    connectedCallback() {  
        const element = this.querySelector('section');  
        element.setAttribute('hidden', '');  
        this.addEventListener('mouseover', () => element.removeAttribute('hidden'));  
        this.addEventListener('mouseout', () => element.setAttribute('hidden', ''));  
    }  
  
}  
  
customElements.define('info-tooltip', InfoToolTipElement);
```

The next example provides the Custom Element for the information of an author:

Online newspaper: Author info

```
const requestCache = {};  
  
class AuthorInfoElement extends HTMLElement {  
  
    async connectedCallback() {  
        const authorId = this.getAttribute('author-id');  
        const request = requestCache[authorId] ||  
            getJSON('/query', {id: generateId(), type: 'GetAuthor', data: {authorId}});  
        requestCache[authorId] = request;  
        const {name, info} = await request;  
        this.innerHTML = `${name}  
            <info-tooltip><a href="#">(i)</a><section>${info}</section></info-tooltip>`;  
    }  
  
}  
  
customElements.define('author-info', AuthorInfoElement);
```

The third Custom Element implementation expresses the concept of an article preview:

Online newspaper: Article Preview

```
class ArticlePreviewElement extends HTMLElement {  
  
    async connectedCallback() {  
        const articleId = this.getAttribute('article-id');  
        const data = awaitgetJSON('/query',  
            {id: generateId(), type: 'GetArticle', data: {articleId}});  
        this.innerHTML = `<h3>${data.title}</h3>  
        <author-info author-id="${data.authorId}"></author-info>  
        <p>${data.summary}</p>`;  
    }  
  
}  
  
customElements.define('article-preview', ArticlePreviewElement);
```

The class `InfoTooltipElement` represents a tooltip element, for which the content must be provided as `<section>` child node. Every other child acts as trigger. The operation `connectedCallback()` initially hides the content and registers mouse event listeners to toggle its visibility. The component `AuthorInfoElement` is responsible for displaying author information. Its function `connectedCallback()` first retrieves the attribute “`author-id`”. Next, it either issues a “`GetAuthorInfo`” Query or uses an existing request from the object `requestCache`. Then, it renders the retrieved author name and an `<info-tooltip>` element for the personal information. The cache mechanism avoids multiple requests for a single author. Finally, the class `ArticlePreviewElement` represents an article preview. Its setup code retrieves an identity, sends a “`GetArticle`” Query and renders the result, including an according `<author-info>` element.



Presentation versus Domain concerns

As explained earlier, UI components can exclusively deal with presentational concerns, but may also incorporate domain-related aspects. The component `InfoTooltipElement` provides a purely presentational functionality. In contrast, both the classes `AuthorInfoElement` and `ArticlePreviewElement` primarily serve a domain-specific purpose.

The next code shows the main file of the client-side JavaScript for the newspaper homepage:

Online newspaper: Rendering latest articles

```
const latestArticlesElement = document.createElement('div');
document.querySelector('body').appendChild(latestArticlesElement);
getJSON('/query', {id: generateId(), type: 'FindLatestArticles'})
.then(articleIds => articleIds.forEach(articleId => {
  const articleElement = document.createElement('article-preview');
  articleElement.setAttribute('article-id', articleId);
  latestArticlesElement.appendChild(articleElement);
}));
```

This is complemented with a basic HTML document that includes the JavaScript code:

Online newspaper: HTML document

```
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Online newspaper</title>
  <link rel="stylesheet" href="/newspaper/ui/index.css">
</head>
<body>
  <script type="module" src="/newspaper/ui/index.js"></script>
</body>
</html>
```

The final code provides the configuration for the newspaper homepage ([run code](#)):

Online newspaper: Configuration and usage

```
const queryHandlers = new NewspaperQueryHandlers();

const queryHttpInterface = createHttpInterface(
  message => queryHandlers.handleQuery(message), ['GET']);

const fsInterface = createHttpFilesystemInterface(({url}) =>
  `${process.cwd()}/..${url === '/' ? '/newspaper/ui/index.html' : url}`);

http.createServer((request, response) =>
  request.url.startsWith('/query') ?
    queryHttpInterface(request, response) : fsInterface(request, response),
).listen(50000);

console.log('<iframe src="http://localhost:50000"></iframe>');
```

The client-side JavaScript code starts with creating and appending a DOM element as container for article previews. Next, it issues the Query “FindLatestArticles” and renders an `<article-preview>` element for each returned identifier. The server-side code starts with instantiating the class `NewspaperQueryHandlers`. Next, an HTTP interface and a filesystem interface are created with the according factory functions. Afterwards, an HTTP server is instantiated. Its listener operation forwards requests with the URL path “/query” to the Query Handlers and all others to the filesystem. The server is started and a message is logged to render an `<iframe>`. Running the code renders a page with all available articles together with their author information. Note that each author is only requested once, independent of the associated article count.

This example illustrates how to use Custom Elements for creating a component model that incorporates both presentation and Domain elements.

Sample Application: User Interface

This section illustrates the implementation of the User Interface for the Sample Application. The goal is to provide a fully functional task board software. On top of the code from previous chapters, each context is complemented with a UI implementation. For serving the according files to the client, a shared HTTP server is operated as standalone program. Overall, the interface design follows a Task-based UI approach. The applied component model uses the Domain Models as conceptual foundation and Custom Elements as technology. Furthermore, the task board part uses Reactive Read Models for real-time updates and collaborative use. As the implementation is primarily for illustration purposes, it must not be understood as feature complete. For example, changing a user’s e-mail address or password is not included.

Shared code

For the implementation of the User Interface, the Sample Application source code is extended with multiple infrastructural components. The operation `createHttpFilesystemInterface()` is introduced for serving files over HTTP. For exposing Query streams over the network, the existing factory `createHttpInterface()` is extended with SSE support. This also requires the introduction of the `ServerSentEventStream` class. The browser-based request helper functions `getJSON()` and `post()` are used for sending Commands and Queries from the client. Also, two stylesheets are added, of which one applies generic styling and the other defines specific rules for task boards. Since the files are purely related to visual aspects, they are considered optional. As a consequence, their contained CSS code is not shown or discussed separately in this section.

User context

The UI implementation for user-related use cases consists of three parts. One is the user creation, which focuses on the initial action that is required to use the software. The second part is the user login, which deals with the authentication of an existing user. The third component is the user profile, which converts a user identity into detail information. For this functionality, the context implementation is extended with a Query type to retrieve a user by its identity. This part is required for the project and the task board context. Both team member Aggregates and task board Aggregates reference user identifiers. With the user profile mechanism, their corresponding UI parts can render user detail information without violating conceptual boundaries.

The first example shows the HTML document for the user creation part:

User context: User creation HTML

```
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Task Board Application | User creation</title>
  <link rel="stylesheet" href="/shared/ui/style.css">
</head>
<body>
  <h1>Task Board Application</h1>
  <user-creation></user-creation>
  <a href="/user/ui/user-login.html">Login</a>
  <script type="module" src="/user/ui/user-creation.js"></script>
</body>
</html>
```

The second code contains the implementation of the user creation Custom Element:

User context: User creation element

```
class UserCreationElement extends HTMLElement {  
  
    connectedCallback() {  
        this.innerHTML = `<form>  
            <input type="text" required name="username" placeholder="Username" />  
            <input type="email" required name="emailAddress" placeholder="Email" />  
            <input type="password" required name="password" placeholder="Password" />  
            <button type="submit">Create user</button>  
        </form>`;  
        this.addEventListener('submit', event => this._handleSubmit(event));  
    }  
  
    async _handleSubmit(event) {  
        event.preventDefault();  
        const {elements: formElements} = event.target;  
        const [username, emailAddress, password] =  
            ['username', 'emailAddress', 'password'].map(name => formElements[name].value);  
        const command = {id: generateId(), type: 'CreateUser', data:  
            {userId: generateId(), username, emailAddress, password, role: 'user'}};  
        const response = await post('/user', command);  
        if (response.status === 200) {  
            sessionStorage.setItem('userId', command.data.userId);  
            document.location.href = '/project/ui/project-overview.html';  
        } else alert(await response.text());  
    }  
  
}  
  
customElements.define('user-creation', UserCreationElement);
```

The class `UserCreationElement` provides the User Interface functionality for creating a user. Its operation `connectedCallback()` renders a `<form>` element and registers a submit event listener. The form contains inputs for a username, an e-mail address and a password. Each of the elements defines the validation attribute `required` to prevent sending invalid Commands. Upon form submission, the handler function `_handleSubmit()` is executed. This operation first extracts the submitted data from the form. Then, it issues an according “CreateUser” command. If the request succeeds, the user identifier is stored in the Session Storage and the project overview page is loaded. In case of an error, the according message is displayed via the function `alert()`. As last step, the Custom Element is registered with the tag name “`user-creation`”.



Sharing data between multiple pages

The **Session Storage** of a browser can be used to share short-lived data across different pages on the same host. Every saved item is retained until the browser is closed. For the Sample Application, this mechanism is used to make the user identity available to all pages and their components.

The following example shows the HTML document for the user login:

User context: User login HTML

```
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Task Board Application | Login</title>
  <link rel="stylesheet" href="/shared/ui/style.css">
</head>
<body>
  <h1>Task Board Application</h1>
  <user-login></user-login>
  <a href="/user/ui/user-creation.html">Create user</a>
  <script type="module" src="/user/ui/user-login.js"></script>
</body>
</html>
```

The next code implements the Custom Element for the user login:

User context: User login element

```
class UserLoginElement extends HTMLElement {

  connectedCallback() {
    this.innerHTML = `<form>
      <input type="email" required name="emailAddress" placeholder="Email" />
      <input type="password" required name="password" placeholder="Password" />
      <button type="submit">Login</button>
    </form>`;
    this.addEventListener('submit', event => this._handleSubmit(event));
  }

  async _handleSubmit(event) {
    event.preventDefault();
    const {elements: form} = event.target;
    const emailAddress = form.emailAddress.value, password = form.password.value;
```

```
const [user] = await getJSON('/user',
  {id: generateId(), type: 'FindUserByEmailAddress', data: {emailAddress}});
if (!user) {
  alert('Login error');
  return;
}
const commandResponse = await post('/user',
  {id: generateId(), type: 'LoginUser', data: {userId: user.id, password}});
if (commandResponse.status === 200) {
  sessionStorage.setItem('userId', user.id);
  location.href = '/project/ui/project-overview.html';
} else alert('Login error');
}

customElements.define('user-login', UserLoginElement);
```

The class `UserLoginElement` is responsible for the user login functionality. Its function `connectedCallback()` renders inputs for an e-mail address and a password, and registers a form submit event listener. The handler operation `_handleSubmit()` first retrieves the submitted data. Next, it issues a “`FindUserByEmailAddress`” Query to determine the user identifier for a given e-mail address. In case of an empty result, an error message is displayed and the process is aborted. In case of success, a “`LoginUser`” Command with the according identity is sent. If the command succeeds, the user identifier is again stored in the Session Storage and the project overview is shown. If an error occurs, a generic alert is provided to the user. The Custom Element is registered with the tag name “`user-login`”.



Eventual consistent process

Logging in a user when provided with an e-mail address and a password is subject to Eventual Consistency. This is because the mapping of an e-mail address to a user identity is achieved by executing the “`FindUserByEmailAddress`” Query. The associated Query handler operation provides the desired information by accessing a Read Model, which is always only eventually consistent.

The final example provides a Custom Element for the user profile mechanism:

User context: User login element

```
const requestCache = {};  
  
const UserProfileElement = BaseElement => class extends BaseElement {  
  
    async connectedCallback() {  
        const id = this.getAttribute('id');  
        const request = requestCache[id] ||  
            getJSON('/user', {id: generateId(), type: 'FindUser', data: {id}});  
        requestCache[id] = request;  
        const user = await request;  
        this.innerHTML = user.username;  
        this.setAttribute('title', user.emailAddress);  
    }  
  
};  
  
customElements.define('user-profile', UserProfileElement(HTMLElement));  
customElements.define('user-profile-option',  
    UserProfileElement(HTMLOptionElement), {extends: 'option'});
```

The code first defines the empty object `requestCache`. Then, the function `UserProfileElement()` is implemented. This operation expects a constructor and returns a Custom Element class that extends the given type. The returned class fetches and renders user details. Its operation `connectedCallback()` first retrieves the “id” attribute value. Next, it either issues a “FindUser” Query or re-uses a cached request. Finally, it renders the username with the e-mail address as “title” attribute. The factory function `UserProfileElement()` is executed twice to create two classes. Both are registered as Custom Elements. While the “user-profile” entry extends the type `HTMLElement`, the “user-profile-option” variant uses the class `HTMLOptionElement` and “option” as `extends` parameter. This enables the user profile to be used both as autonomous element and as select option.

Project context

The User Interface related to the project context consists of two parts that are implemented as three components. One is the project overview, which is responsible for creating new projects and rendering existing ones. This includes both projects that are owned by a user and the ones a user collaborates in. The second component is the team member creation, which provides the ability to add new team members to a team. This component interacts

with the user context for translating an e-mail address into a user identity. Viewing the list of existing members and removing individual ones is enabled through the team member list component. For this functionality, the context implementation is extended with a Query type to retrieve the members of a team.

The first example shows the HTML document for the project overview:

Project context: Project overview HTML

```
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Task Board Application | Project Overview</title>
  <link rel="stylesheet" href="/shared/ui/style.css">
</head>
<body>
  <h1>Project overview</h1>
  <project-overview></project-overview>
  <script type="module" src="/project/ui/project-overview.js"></script>
</body>
</html>
```

The next code implements the Custom Element for the project overview:

Project context: Project overview element

```
class ProjectOverviewElement extends HTMLElement {

  async connectedCallback() {
    this.innerHTML = `<form>
      <input type="text" required name="name" placeholder="Project name">
      <button type="submit">Create project</button>
    </form>
    <h3>Projects you own</h3><ul class="owned-projects"></ul>
    <h3>Projects you collaborate in</h3><ul class="collaborating-projects"></ul>`;
    this.addEventListener('submit', event => this._handleSubmit(event));
    const data = {userId: sessionStorage.getItem('userId')};
   getJSON('/project', {type: 'FindProjectsByOwner', data})
      .then(projects => this._addProjects('owned-projects', projects));
   getJSON('/project', {type: 'FindProjectsByCollaboratingUser', data})
      .then(projects => this._addProjects('collaborating-projects', projects));
  }

  _addProjects(projectType, projects) {
```

```

const template = document.createElement('template');
template.innerHTML = projects.map(({name, teamId, taskBoardId}) => `<li>
  ${name}
  <a href="/project/ui/team-management.html?id=${teamId}">Team</a>,
  <a href="/task-board/ui/task-board.html?id=${taskBoardId}&team-id=${teamId}">
    Task Board
  </a>
</li>`).join('');
this.querySelector(`.${projectType}`).appendChild(template.content);
}

async _handleSubmit(event) {
  event.preventDefault();
  const name = event.target.elements.name.value;
  const ownerId = sessionStorage.getItem('userId');
  const id = generateId(), teamId = generateId(), taskBoardId = generateId();
  const project = {projectId: id, name, ownerId, teamId, taskBoardId};
  const projectCommand = {id: generateId(), type: 'CreateProject', data: project};
  const response = await post('/project', projectCommand);
  if (response.status === 200) this._addProjects('owned-projects', [project]);
  else alert('Project creation failed.');
}

customElements.define('project-overview', ProjectOverviewElement);

```

The class `ProjectOverviewElement` provides the User Interface for creating new projects and viewing existing ones. Its operation `connectedCallback()` first renders a form and two list containers, one for owned projects and one for collaborations. Next, a form submit event listener is registered. Then, the user identity is retrieved from the Session Storage. Afterwards, the Queries “`FindProjectsByOwner`” and “`FindProjectsByCollaboratingUser`” are issued. For each response, the operation `_addProjects()` is invoked together with the according project type. This function creates a `<template>` element, renders every project as list item and appends the result to the according container. The operation `_handleSubmit()` retrieves submitted data, issues a “`CreateProject`” command and either updates the project list or shows an error. For the Custom Elements registry entry, the name “`team-member-list`” is used.

The following example shows the HTML document for the team management:

Project context: Team management HTML

```
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Task Board Application | Team Management</title>
  <link rel="stylesheet" href="/shared/ui/style.css">
</head>
<body>
  <h1>Team Management</h1>
  <aside>
    <a href=".//project-overview.html">&lt;&lt; Back to project overview</a>
  </aside>
  <team-member-creation></team-member-creation>
  <team-member-list></team-member-list>
  <script type="module" src="/project/ui/team-member-creation.js"></script>
  <script type="module" src="/project/ui/team-member-list.js"></script>
</body>
</html>
```

The first team management part is the Custom Element for the team member creation:

Project context: Team member creation element

```
class TeamMemberCreationElement extends HTMLElement {

  connectedCallback() {
    this._teamId = new URLSearchParams(location.search).get('id');
    this.innerHTML = `<form>
      <input type="text" required name="emailAddress" placeholder="E-Mail address">
      <input type="text" required name="role" placeholder="Role">
      <button type="submit">Add team member</button>
    </form>`;
    this.addEventListener('submit', event => this._handleSubmit(event));
  }

  async _handleSubmit(event) {
    event.preventDefault();
    const {elements} = event.target;
    const emailAddress = elements.emailAddress.value, role = elements.role.value;
    const [user] = await getJSON('/user',
      {id: generateId(), type: 'FindUserByEmailAddress', data: {emailAddress}});
    if (!user) {
      alert('User not found');
```

```
    return;
}
const {status} = await post('/project',
  {id: generateId(), type: 'AddTeamMemberToTeam', data:
   {teamMemberId: generateId(), teamId: this._teamId, userId: user.id, role}});
if (status === 200) window.location.reload();
else alert('Team member creation failed.');
}

customElements.define('team-member-creation', TeamMemberCreationElement);
```

The type `TeamMemberCreationElement` is responsible for the functionality of adding a new team member to a team. Its setup code starts with retrieving the team identifier from a URL query parameter. Afterwards, it renders a form and registers a submit event listener. The handler operation `_handleSubmit()` starts with retrieving submitted data. Then, similar to the user login, it issues a “`FindUserByEmailAddress`” Query to map an e-mail address to a user identity. Next, it sends an “`AddTeamMemberToTeam`” Command. If the request succeeds, the page is reloaded using the function `window.location.reload()`. This is done to ensure that the team member list component is updated automatically. In case of an error, the associated message is shown. As last step, the Custom Element is registered with the name “`team-member-creation`”.



Communication with foreign contexts

Sending a Query from a User Interface part of one context to the Application Layer of another one is acceptable. Nevertheless, implementing a dedicated component in the owning context for the required functionality can provide a better encapsulation of responsibilities.

The second team management part is the Custom Element for the team member list:

Project context: Team member list element

```
class TeamMemberListElement extends HTMLElement {  
  
    connectedCallback() {  
        this._teamId = new URLSearchParams(location.search).get('id');  
        this.innerHTML = '<h2>Team Members</h2><ul class="team-members"></ul>';  
        this.addEventListener('click', event => this._handleRemoveButtonClick(event));  
        getJSON('/project', {type: 'FindTeamMembers', data: {teamId: this._teamId}})  
            .then(members => this._addTeamMembers(members));  
    }  
  
    _addTeamMembers(members) {  
        const template = document.createElement('template');  
        template.innerHTML = members.map(({id, userId}) => `<li>  
            <user-profile id=${userId}></user-profile>  
            <button class="small" id="${id}">X</button>  
        </li>`).join('');  
        this.querySelector('.team-members').appendChild(template.content);  
    }  
  
    async _handleRemoveButtonClick({target}) {  
        if (!target.matches('button[id]')) return;  
        const data = {teamMemberId: target.getAttribute('id'), teamId: this._teamId};  
        const {status} = await post('/project',  
            {id: generateId(), type: 'RemoveTeamMemberFromTeam', data});  
        if (status === 200) target.closest('li').remove();  
        else alert('Team member removal failed');  
    }  
  
}  
  
customElements.define('team-member-list', TeamMemberListElement);
```

The class `TeamMemberListElement` enables to view and delete members of a team. Its operation `connectedCallback()` also first retrieves the team identity from the URL. Then, it renders a list container and registers a click event listener. Afterwards, it issues a “FindTeamMembers” Query and executes the function `_addTeamMembers()` with the response data. This operation creates a `<template>` element, renders an entry for every received member and appends the result to the list. Each entry consists of a `<user-profile>` element with the according identifier and a remove button. The handler operation `_handleRemoveButtonClick()` first verifies that the respective event occurred on a remove button. Then, it sends a “Re-

moveTeamMemberFromTeam” Command and either updates the list or reports the error. Finally, the Custom Element is registered as “team-member-list”.

Task board context

The User Interface implementation for the task board part consists of three separate components. One is the task creation, which provides the functionality for adding a new task to a board. The second component is the task item, which represents an individual task. This element is used both for displaying information inside a task board and accepting data modifications to request Command executions. The third component is the task board, which renders tasks inside board columns and reacts to Read Model updates. For the Reactive Read Model behavior, the context implementation is adjusted. The projection component is extended to publish change notifications over a dedicated Message Bus. Also, the Query type “StreamTasksOnTaskBoard” is introduced together with a handler operation that yields a Query Stream.

The first example shows an excerpt of the extended Read Model projection:

Task board context: Read Model projection

```
class TaskBoardReadModelProjection {  
  
    #taskReadModelMessageBus; #taskReadModelStorage;  
  
    constructor({eventStore, taskReadModelMessageBus, taskReadModelStorage}) {  
        this.#taskReadModelMessageBus = taskReadModelMessageBus;  
        this.#taskReadModelStorage = taskReadModelStorage;  
        const eventTypesToHandle = [TaskCreatedEvent, TaskAddedToTaskBoardEvent,  
            TaskRemovedFromTaskBoardEvent, TaskTitleChangedEvent,  
            TaskDescriptionChangedEvent, TaskAssigneeChangedEvent,  
            TaskStatusChangedEvent].map(Event => Event.type);  
        this.activate = () =>  
            eventStore.subscribe('$global', event => {  
                if (eventTypesToHandle.includes(event.type)) this.handleEvent(event);  
            });  
    }  
  
    handleEvent = createMessageForwarder(this, {messageSuffix: 'Event'});  
  
    async handleTaskCreatedEvent({data}) {  
        const {taskId, title, description, status, assigneeId} = data;  
        const updates = {id: taskId, title, description, status, assigneeId};  
    }  
}
```

```

    await this.#taskReadModelStorage.update(taskId, updates);
    const task = await this.#taskReadModelStorage.load(taskId);
    if (task.taskBoardId) this.#publishChangeMessage(task.taskBoardId, [task]);
}

/* ... */

#publishChangeMessage(taskBoardId, tasks) {
    this.#taskReadModelMessageBus.publish(`TaskBoard/${taskBoardId}`, tasks);
}

}

```

The next code shows the implementation of the according Query Handler component:

Task board context: Query Handlers

```

class TaskBoardQueryHandlers {

    #taskReadModelStorage; #taskReadModelMessageBus;

    constructor({taskReadModelStorage, taskReadModelMessageBus}) {
        this.#taskReadModelStorage = taskReadModelStorage;
        this.#taskReadModelMessageBus = taskReadModelMessageBus;
        this.handleQuery = createMessageForwarder(this, {messageSuffix: 'Query'});
    }

    async handleStreamTasksOnTaskBoardQuery({data: {taskBoardId}}) {
        const taskBoardStream = new Readable({objectMode: true, read() {}});
        taskBoardStream.push(
            await this.#taskReadModelStorage.findIndex('taskBoardId', taskBoardId));
        this.#taskReadModelMessageBus.subscribe(`TaskBoard/${taskBoardId}`,
            changedTasks => taskBoardStream.push(changedTasks));
        return taskBoardStream;
    }
}

```

The reworked component `TaskBoardReadModelProjection` publishes change notifications after each Read Model update. Its constructor expects a Message Bus as additional argument. Each event handler operation is extended with the publishing of a change notification. The corresponding task board identifier is used as message topic and the data contains a list

with the modified task. For consuming a Reactive Read Model, the specialized Query type “StreamTasksOnTaskBoard” is introduced. In fact, it replaces the previously existing type “FindTasksOnTaskBoard”. The Query Handlers component is adjusted accordingly. For one, its constructor also expects a Message Bus instance. Secondly, the handler for “FindTasksOnTaskBoard” Queries is replaced with the function `handleStreamTasksOnTaskBoardQuery()`. The operation instantiates a stream, transmits the current tasks for a board, configures change notification forwarding and returns the stream.



Tasks without task board identifier

When executing a “RemoveTaskFromTaskBoard” Command, the according task board identifier is removed from the affected task Read Model. Still, a change notification with the modified task as data is published under the topic of the former board identity. This enables the consumers of a Query Stream for a specific board to react to task removals.

The next example provides the HTML document for the task board:

Task board context: Task board HTML

```
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Task Board Application | Task Board</title>
  <link rel="stylesheet" href="/shared/ui/style.css">
  <link rel="stylesheet" href="/task-board/ui/style.css">
</head>
<body>
  <h1>Task Board</h1>
  <task-creation></task-creation>
  <task-board></task-board>
  <script type="module" src="/task-board/ui/task-creation.js"></script>
  <script type="module" src="/task-board/ui/task-board.js"></script>
  <script type="module" src="/task-board/ui/task-item.js"></script>
  <script type="module" src="/user/ui/user-profile.js"></script>
</body>
</html>
```

This is followed by the code for the Custom Element to create a task:

Task board context: Task creation element

```
class TaskCreationElement extends HTMLElement {

    async connectedCallback() {
        const queryParameters = new URLSearchParams(location.search);
        this._taskBoardId = queryParameters.get('id');
        this.innerHTML = `<form>
            <input required type="text" name="title" placeholder="Title">
            <input type="text" name="description" placeholder="Description">
            <button type="submit">Create task</button>
        </form>`;
        this.addEventListener('submit', event => this._handleSubmit(event));
    }

    async _handleSubmit(event) {
        event.preventDefault();
        const [title, description] =
            ['title', 'description'].map(name => event.target.elements[name].value);
        const id = generateId(), taskId = generateId();
        const {status} = await post('/task-board', {id, type: 'AddNewTaskToTaskBoard',
            data: {taskId, title, description, taskBoardId: this._taskBoardId}});
        if (status !== 200) alert('Creation failed.');
    }
}

customElements.define('task-creation', TaskCreationElement);
```

The class `TaskCreationElement` enables to add a new task to a task board. Its operation `connectedCallback()` retrieves the task board identifier from the URL, renders a form and registers a submit event listener. The `<form>` element exclusively contains inputs for a title and a task description. Note that the Application Service for creating a task allows to provide more information, such as status and assignee identifier. Here, the interface design aims for an improved user experience through less form inputs. After a task creation, the fields can be updated separately. The handler function `_handleSubmit()` retrieves submitted data and issues an according “`AddNewTaskToTaskBoard`” command. If the request fails, an error message is displayed via the function `alert()`. The Custom Element is registered with the name “`task-creation`”.

The next code provides the implementation for the task item component:

Task board context: Task item element

```
class TaskElement extends HTMLElement {  
  
    async connectedCallback() {  
        if (this.children.length > 0) return;  
        const statusOptions = ['todo', 'in progress', 'done'].map(  
            status => `<option name="${status}" value="${status}">${status}</option>`);  
        this.innerHTML = `<form><strong class="task-id">Task ID: ${this.id}</strong>  
        <button class="remove small" type="button">Remove</button>  
        <textarea type="text" required name="title" placeholder="Title"></textarea>  
        <textarea type="text" name="description" placeholder="Description"></textarea>  
        <select name="status">${statusOptions}</select>  
        <select name="assigneeId"></select></form>`;  
        this.addEventListener('change', event => this._handleFormInteraction(event));  
        this.querySelector('.remove').addEventListener('click', () => this._remove());  
        this.addEventListener('click', ({target}) => {  
            this.classList[target === this ? 'remove' : 'add']('large-view');  
        });  
    }  
  
    setTeamMembers(teamMembers) {  
        this.querySelector('[name=assigneeId]').innerHTML =  
            `<option name="none" value="">unassigned</option>\n` +  
            `${teamMembers.map(({id, userId}) => `<option is="user-profile-option"  
                id="${userId}" name="${id}" value="${id}"></option>`).join('')}`;  
    }  
  
    updateData(data) {  
        const formElements = this.querySelector('form').elements;  
        for (const [key, value] of Object.entries(data)) {  
            const element = formElements[key];  
            if (!element) continue;  
            if (element.options) element.options[value || 'none'].selected = true;  
            else element.value = value;  
        }  
        const isAssigneeDisabled = data.status !== 'todo';  
        this.querySelector('[name=assigneeId] option').disabled = isAssigneeDisabled;  
        this.querySelector('[name=status]').disabled = !data.assigneeId;  
    }  
  
    async _handleFormInteraction(event) {  
        const formElement = event.target.closest('select, textarea');  
        if (!formElement) return;  
    }  
}
```

```
const attribute = formElement.getAttribute('name'), value = formElement.value;
const commandSuffixByAttribute = {title: 'Title', description: 'Description',
    status: 'Status', assigneeId: 'Assignee'};
const type = `UpdateTask${commandSuffixByAttribute[attribute]}`;
const {status} = await post('/task-board', {id: generateId(), type,
    data: {taskId: this.getAttribute('id'), [attribute]: value}});
if (status !== 200) alert('Task update failed, please reload page.');
}

async _remove() {
    const data = {taskId: this.getAttribute('id'),
        taskBoardId: this.getAttribute('task-board-id')};
    await post('/task-board',
        {id: generateId(), type: 'RemoveTaskFromTaskBoard', data});
}

customElements.define('task-item', TaskElement);
```

The component `TaskElement` provides the User Interface for an individual task. Its setup code renders a `<form>` element and registers multiple event listeners. The class defines two functions for injecting data from the outside. Setting the list of team members as possible assignees is achieved via the operation `setTeamMembers()`. The function `updateData()` expects task data for populating the contained form elements. The handler operation `_handleFormInteraction()` is executed whenever the value of a form input element changes. Depending on the name of the modified input, an according Command is created and sent. If the request fails, an error message is displayed. Finally, the handler operation `_remove()` is responsible for issuing a “`RemoveTaskFromTaskBoard`” Command. For the Custom Elements registry entry, the name “`task-item`” is used.



View and edit mode for tasks

The task item component supports displaying and editing information. For both modes, the according stylesheet defines separate rules. When clicking on the content area of a task item, it enters the editing mode and shows a backdrop element. When clicking on the backdrop, the task switches back to the view mode.

The last example shows the Custom Element for the task board:

Task board context: Task board element

```
class TaskBoardElement extends HTMLElement {

    async connectedCallback() {
        this.innerHTML = ['todo', 'in progress', 'done'].map(
            status => `<ul class="task-column" data-status="${status}"></ul>`).join('');
        const parameters = new URLSearchParams(location.search);
        const query =
            {type: 'StreamTasksOnTaskBoard', data: {taskBoardId: parameters.get('id')}};
        const eventSource = new EventSource(`/task-board?${createQueryString(query)}`);
        eventSource.addEventListener(
            'message', (data) => this._addOrUpdateTasks(JSON.parse(data)));
        this._teamMemberRequest = getJSON('/project',
            {type: 'FindTeamMembers', data: {teamId: parameters.get('team-id')}});
    }

    async _addOrUpdateTasks(tasks) {
        const teamMembers = await this._teamMemberRequest;
        tasks.forEach(task => {
            let taskItem = this.querySelector(`[id="${task.id}"]`);
            if (!task.taskBoardId) {
                taskItem.remove();
                return;
            }
            if (!taskItem) {
                taskItem = document.createElement('task-item');
                taskItem.setAttribute('id', task.id);
                taskItem.setAttribute('task-board-id', task.taskBoardId);
            }
            const columnElement = this.querySelector(`[data-status='${task.status}']`);
            if (!columnElement.querySelector(`[id='${task.id}']`))
                columnElement.appendChild(taskItem);
            taskItem.setTeamMembers(teamMembers);
            taskItem.updateData(task);
        });
    }

    customElements.define('task-board', TaskBoardElement);
}
```

The class `TaskBoardElement` is responsible for displaying board columns and tasks. Its

function `connectedCallback()` first renders the column elements. Then, it creates a “Stream-TasksOnTaskBoard” Query. Next, the class `EventSource` is instantiated with the serialized Query as URL parameters and a message listener is registered. Finally, a “FindTeamMembers” Query is issued and the request is saved. The handler operation `_addOrUpdateTasks()` starts with awaiting the team member request. Then, it iterates the given task collection. If a task has no board identifier, its associated element is removed. Otherwise, if no element is existing, it is created. Also, the element is ensured to be contained in the correct column. Finally, the task item functions `setTeamMembers()` and `updateData()` are executed. The Custom Element is registered with the name “task-board”.

User Interface server

The previous defined program layout is extended with a standalone HTTP file server for the User Interface. This component serves the required client files from all three contexts as well as the shared directory. As with the other programs, an HTTP proxy is used that forwards requests to the file server. This allows a user to communicate with a single server that provides both the User Interface and all Application Services. The custom proxy resolver mechanism is adjusted accordingly. Every request with a file ending in its URL is forwarded to the file server. Also, the default URL path “/” is forwarded to the HTML document for the user login page. All other requests are assumed to target an Application Service component.

The last example shows the final setup code to run the Sample Application ([run code](#)):

Sample Application: Setup code

```
const programMappings = {
  'user/write-side': 50001,
  'user/read-side': 50002,
  'project/write-side': 50003,
  'project/read-side': 50004,
  'task-board/write-side': 50005,
  'task-board/read-side': 50006,
};

Object.entries(programMappings).forEach(([program, PORT]) =>
  spawnedProcesses.push(childProcess.spawn('node',
    [path.join(rootDirectory, program)],
    {env: {...process.env, ROOT_STORAGE_DIR, PORT}, stdio: 'inherit'})));
}

const fileServerPort = 50007;
const fsInterface = createHttpFilesystemInterface(request => {
  const {pathname: urlPath} = parse(request.url);
```

```
return path.join(
  rootDirectory, urlPath === '/' ? '/user/ui/user-login.html' : urlPath);
});
http.createServer(fsInterface).listen(fileServerPort);

const httpProxy = createHttpProxy(({method, url}) => {
  const {pathname, query} = parse(url);
  const program = `${pathname.substr(1)}/${method === 'POST' ? 'write' : 'read'}-side`;
  return programMappings[program]
    ? `http://localhost:${programMappings[program]}${query}`
    : `http://localhost:${fileServerPort}${pathname}${query ? `?${query}` : ''}`;
});

http.createServer(httpProxy).listen(50000);

console.log('<iframe src="http://localhost:50000"></iframe>');


---


```



Using a separate browser window

After executing the Sample Application setup code in the Playground, the software can be accessed in a separate window. This is because the spawned Node.js process is not terminated and the software continues to operate.

The User Interface implementation extends the Sample Application into a fully functional task board software. Each of the three contexts provide their own UI parts as Custom Elements. The shared HTTP file server in combination with the proxy allows a unified access to all relevant files. Finally, the use of Reactive Read Models enable a collaborative task board usage.

Conclusion

The concepts covered in this book enable to build software in a distinct way. **Domains**, **Domain Models** and **Bounded Contexts** put emphasis on the problem space, on knowledge abstractions and on conceptual boundaries. The **Layered Architecture** and the **Onion Architecture** establish a modular software structure. **Value Objects**, **Entities** and **Domain Services** drive the design of Domain Model implementations. **Domain Events** express important occurrences and facilitate message exchange. **Aggregates** define transactional consistency boundaries. **Repositories** implement persistence in a meaningful way to a Domain. **Application Services** manage use case executions. **CQRS** separates write concerns and read concerns. **Event Sourcing** represents state as a set of transitions and emphasizes behavior. **Program separation** allows autonomy and scaling. **Task-based UIs** with **Reactive Read Models** empower User Interfaces with real-time capabilities.

Apply the useful parts

Every concept and pattern described in this book should only be applied when it makes sense to do so. The design of abstractions, architectures and implementations must be driven by the desire to solve a problem with software. Concepts and technologies are merely means to an end. Their incorrect usage can introduce unnecessary complexity and in the worst case cause a project to fail. Especially CQRS and Event Sourcing should only be applied to suitable areas of a software. Of course, it is also possible to utilize a subset of the covered concepts. For example, many of the DDD patterns can be used independently of other parts. Similarly, CQRS and Event Sourcing can be implemented on their own.

The use of frameworks

There are many frameworks for implementing software with CQRS and Event Sourcing. In fact, I worked on one such project myself. Generally speaking, I would not recommend their use and often find them problematic. There are multiple reasons for this. For one, some frameworks are very opinionated and strict about the way CQRS and Event Sourcing is applied. They enforce a certain implementation style and the use of specific technical

constructs. In the worst case, they infiltrate most aspects of a software and cause a vendor lock-in. Another issue is that some frameworks work well for simple cases, but fail to handle more specialized and complex scenarios. However, note that this is only my personal opinion and not an objective or universal recommendation.

Appendix A: Static types

Static types enforce type constraints at compile-time before the execution of a program. They can help to improve correctness and lower the chances of runtime errors, which is often otherwise achieved through testing. Also, they can enable to express domain-specific characteristics and relationships on a type-level. Therefore, the use of static types can be particularly beneficial for the Domain Layer of a software. JavaScript itself is an untyped language. However, there are statically typed languages that can compile to JavaScript, such as TypeScript. This appendix discusses static typing for selected concepts from the book and illustrates them with TypeScript examples. Wherever it makes sense, the original example topics are re-used. For the Sample Application, the included source code bundle provides a full TypeScript-based implementation.



Why not TypeScript for the whole book?

The main part of this book uses plain JavaScript due to multiple reasons. For one, JavaScript is more widespread than TypeScript and has a lower entry barrier. Secondly, the absence of type annotations ensures to keep the code examples compact. Also, while static typing is often helpful for a Domain Model implementation, other architectural parts may benefit less from it.

Value Objects, Entities and Services

Static types extend and improve the possibilities how Value Objects, Entities and Domain Services can be expressed as code. Independent of the Domain Layer, there are some universal positive aspects. As mentioned previously, static types improve the overall code correctness. Also, they commonly serve as replacement for custom runtime type-checks. With regard to a Domain Model implementation, there are some further specific benefits. On a component level, individual attributes and behaviors can be defined more precisely through suitable type annotations. On a composite level, component relationships can be described with custom types and interfaces. This can help to create a more readable Domain Model expression. Even more, it can contribute to the goal of making the code an artifact that can be read by non-developers.



Executing the TypeScript code

The Code Playground supports the compilation and the execution of TypeScript examples. The only additional requirement is that the TypeScript compiler is installed globally. This can be done by executing `npm i -g typescript`. Note that compilation errors are ignored and not logged to the output. Instead, individual issues can be inspected in the code editor by hovering over them.

The following code shows a statically typed implementation of the [address book example](#) from Chapter 6 ([run code](#)):

Address Book: Static types

```
class Name {  
  
    firstName: string; lastName: string;  
  
    constructor({firstName, lastName}: {firstName: string, lastName: string}) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
        Object.freeze(this);  
    }  
  
}  
  
class Address {  
  
    street: string; houseNumber: string; zipCode: string; country: string;  
  
    constructor({street, houseNumber, zipCode, country}: {  
        street: string, houseNumber: string, zipCode: string, country: string  
    }) {  
        this.street = street;  
        this.houseNumber = houseNumber;  
        this.zipCode = zipCode;  
        this.country = country;  
        Object.freeze(this);  
    }  
  
}  
  
class Contact {  
  
    id: string; name: Name; address: Address;
```

```
constructor({id, name, address}: {id: string, name: Name, address: Address}) {
    this.id = id;
    this.name = name;
    this.address = address;
    Object.defineProperty(this, 'id', {writable: false});
}

const contact1 = new Contact({id: generateId(),
    name: new Name({firstName: 'John', lastName: 'Doe'}),
    address: new Address({street: 'First Street',
        houseNumber: '4', zipCode: '19061', country: 'Germany'})});
console.log(contact1);
```

The Domain Model implementation is more intention-revealing and more expressive compared to its untyped counterpart. While the classes `Name` and `Address` represent the Value Object types of the Domain Model, the component `Contact` is its main Entity type. All attributes of both Value Object classes are annotated with the type `string`. This may seem overly generic and partially even incorrect. However, there are two specific fields that benefit in particular from the type annotations. One common misconception with postal addresses is to assume that house numbers and zip codes are numerical values. In fact, both attributes are best represented as strings, as they can contain alphanumerical values. Finally, the contact Entity type expresses its component relationships in a more expressive and type-safe way.

Immutability

The immutability of individual attributes and complete components can be enforced through runtime behavior or through static types. In TypeScript, the modifier `readonly` marks an attribute as immutable. Both approaches have their own advantages and implications. Defining immutability using a static type system prevents the execution of code that attempts to perform an invalid modification. However, in order for this to work correctly, every consumer of the code must perform static type checking. Enforcing immutability with runtime behavior through mechanisms such as `Object.freeze()` makes modifications impossible. The disadvantage of this approach is that violation attempts are only discovered during execution and may trigger exceptions. Theoretically, both approaches can be combined. Though, when using static types, defining immutability exclusively on a type-level is normally sufficient.

The following implementation of the address book example replaces the execution of `Object.freeze()` and `Object.defineProperty()` with type-level immutability:

Address Book: Immutability on type-level

```
class Name {  
    readonly firstName: string; readonly lastName: string;  
  
    constructor({firstName, lastName}: {firstName: string, lastName: string}) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
}  
  
class Address {  
    readonly street: string; readonly houseNumber: string;  
    readonly zipCode: string; readonly country: string;  
  
    constructor({street, houseNumber, zipCode, country}: {  
        street: string, houseNumber: string, zipCode: string, country: string}) {  
        this.street = street;  
        this.houseNumber = houseNumber;  
        this.zipCode = zipCode;  
        this.country = country;  
    }  
}  
  
class Contact {  
    readonly id: string; name: Name; address: Address;  
  
    constructor({id, name, address}: {id: string, name: Name, address: Address}) {  
        this.id = id;  
        this.name = name;  
        this.address = address;  
    }  
}
```

Value Object as types

Value Objects that have no behavior on their own can be exclusively defined as types without implementing class components. The types are best understood as structural contracts that need to be satisfied, regardless of creation mechanisms. Whether an instance is created via object literals, factories or classes is irrelevant, as long as it matches the type. When using TypeScript, this approach must be chosen with care. As explained earlier, TypeScript is a language that compiles to JavaScript. For the most parts, the compilation step strips all type information from the code that is later executed. When defining a Value Object component exclusively as static type, there is no notion of it at runtime. Depending on the implementation approach, this can be either desired or not.

The final implementation of the address book example defines the Value Object components exclusively as static types:

Address Book: Value Objects as types

```
type Name = {
    readonly firstName: string;
    readonly lastName: string;
};

type Address = {
    readonly street: string;
    readonly houseNumber: string;
    readonly zipCode: string;
    readonly country: string;
};

class Contact {

    readonly id: string; name: Name; address: Address;

    constructor({id, name, address}: {id: string, name: Name, address: Address}) {
        this.id = id;
        this.name = name;
        this.address = address;
    }
}

const contact1 = new Contact({id: generateId(),
    name: {firstName: 'John', lastName: 'Doe'},
```

```
address: {street: 'First Street',
  houseNumber: '4', zipCode: '19061', country: 'Germany'}});
contact1.address = {...contact1.address, street: 'First Street', houseNumber: '16'};
console.log(contact1);
```

Invariants

Whenever possible, invariants should be expressed on a type-level. The feasibility of this approach depends on the specific constraints and the capabilities of the type system. As described in [Chapter 6](#), an invariant is “about protecting components from entering an invalid state”. This protection should happen at the earliest possible point in time. While it is acceptable and sometimes inevitable to perform runtime checks, type-level constraints can be superior. When all logical invariants are expressed through static types, there is no possibility for an unexpected violation attempt during runtime. However, the type systems of many programming languages lack the necessary capabilities to express complex domain-specific constraints. While TypeScript has a powerful type system, it can also only cover limited scenarios for invariant protection.

Example: Survey engine

Consider implementing the Domain Model of a survey engine. The goal is to provide a possibility for conducting surveys and evaluating their results. This example exclusively focuses on a simplified version of a question component. The concept is defined as an immutable combination of a question text and possible answers. Therefore, it qualifies as Value Object. The question text is a plain string, while the possible answers are a collection of strings. There are two invariants involved in the selected Domain Model part. For one, the question text must not exceed a length of 150 characters. The second constraint is that the possible answers must at least contain two entries. These invariants must be protected at all times, regardless of technological capabilities and implementation approaches.

The following code shows one possible implementation of the question Value Object component ([run code](#)):

Survey Engine: Question Value Object

```
type PossibleAnswers = Array<string> & {0: string, 1: string};

class Question {

    question: string;
    possibleAnswers: PossibleAnswers;

    constructor(question: string, possibleAnswers: PossibleAnswers) {
        if (question.length > 150) throw new Error('question is too long');
        this.question = question;
        this.possibleAnswers = possibleAnswers;
    }
}

const question1 = new Question('Do you like static types?', ['Maybe']);
```

The code implements the question concept together with its associated invariants. One of them is protected through static typing, the other one via a runtime check. The type `PossibleAnswers` expresses the invariant minimum count of possible answers. This is done by combining a string array type and a type with strings at index 0 and 1. Simply put, the type represents a string array with a minimum length of 2. The class `Question` expects a question text and the possible answers as constructor arguments. For protecting the invariant maximum question length, a runtime check is performed. Another feasible implementation approach is to also enforce the minimum count of answers at runtime. In contrast, protecting both invariants through static types is not possible with TypeScript.

Events

Events are messages that are typically exchanged between software parts. Their purpose is to convey structured knowledge. In most cases, the data structure should be homogenous across instances of the same type. This is equally true for Domain Events that are published outside their conceptual boundary and for Event Sourcing records. The [event creation mechanism](#) described in Chapter 7 incorporates multiple responsibilities within a single factory. One part is the enforcement of a data structure upon event instantiation. This aspect can be completely replaced with the use of static types. The second part is a configurable process of creating metadata without affecting the Domain Layer. Other than the data structure aspect, this functionality is a runtime behavior that must exist independent of static typing.

The following code shows a TypeScript version of the event type factory from Chapter 7:

Events: Event Type Factory

```
type IdGenerator = () => string;
type Metadata = Record<string, object>;
type MetadataProvider = () => Metadata;

let generateId: IdGenerator = () => '';
let createMetadata: MetadataProvider = () => ({});

const createEventType = <TypeName extends string>(type: TypeName) => <Data>() => {
    return class Event {

        static readonly type: TypeName = type;
        readonly type = type;
        readonly id = generateId();
        readonly metadata = createMetadata();

        constructor(readonly data: Data) {}

    }
}

const setIdGenerator = (idGenerator: IdGenerator) => generateId = idGenerator;

const setMetadataProvider =
    (metadataProvider: MetadataProvider) => createMetadata = metadataProvider;

const eventTypeFactory = {createEventType, setIdGenerator, setMetadataProvider};
```

The factory function `createEventType()` enables to define event types with statically type-checked data and automatically generated metadata. Conceptually, it has the same responsibilities as the JavaScript implementation. The main difference in its signature is that it expects the data structure as type argument. Other than the JavaScript variant, the factory defines a class instead of a plain constructor function. This is because TypeScript has a better support for the class syntax. The behavior of custom event types remains almost unchanged, only that invalid data triggers compilation errors instead of runtime exceptions. The functions `setIdGenerator()` and `setMetadataProvider()` exclusively differ in the type annotations for their arguments. In contrast to the data of an event, the metadata part is less strictly typed and can contain arbitrary properties.



No partial type inference

The TypeScript implementation of the event type factory uses two type parameters, of which one can be inferred. However, at the time of writing, TypeScript does not support partial type inference. For this reason, the factory is implemented as a sequence of two functions. While the first operation infers the type parameter, the second one expects an explicit definition.

Differentiating event types

The event type name is defined as generic type so that TypeScript can infer event types based on it. This is possible due to String Literal Types and Discriminating Unions. String literals are of type `string`, but also have a literal type that is equal to their value. When a string function argument is generic, every invocation with a string literal uses its literal type. This way, events of separate types have a different type for their type name field, not only a different value. For multiple types that share a field but with individual types, TypeScript can differentiate between them by evaluating runtime checks. When a code branch assumes an event type name, its literal type and its according event type can be inferred.

The following code illustrates a usage of the event type factory and the differentiation of event types by type name ([run code](#)):

Events: Event Type Factory usage

```
const CommentWrittenEvent = createEventType('CommentWritten')
  <{commentId: string, message: string, author?: string}>();
const CommentDeletedEvent = createEventType('CommentDeleted')
  <{commentId: string}>();

const commentId = generateId();

const event1 = new CommentWrittenEvent(
  {commentId, message: 'Very nice content!', author: 'John'});
const event2 = new CommentDeletedEvent({commentId});

const events = [event1, event2];

events.forEach(event => {
  if (event.type == CommentWrittenEvent.type)
    console.log(` ${event.data.author}: ${event.data.message}`);
});
```

The example first defines the event types `CommentWrittenEvent` and `CommentDeletedEvent`. Then, two event instances are created, one for each type. These events do not only have a different value for their type name, but also a different type. Next, they are gathered in a collection, of which the item type is combination of the two event types. While the type name is a string with two possible values, the data is an object with two possible structures. When asserting a specific type name, TypeScript can infer its literal type and the event type. Therefore, when the code checks for the type name “`CommentWritten`”, TypeScript knows that the event type is `CommentWrittenEvent`. Consequently, the data part is known to contain a message and an optional author.

Event-sourced Write Model

The use of static types influences an event-sourced Write Model in a similar way as it influences a fixed-state-based implementation. Therefore, the beneficial aspects for Value Objects and Entities described in the previous section also apply in this context. In general, static types help to achieve an improved code correctness. For certain domain-specific constraints, they eliminate the need for custom runtime type-checks. Also, their use enables to produce intention-revealing, expressive and readable source code. As an event-sourced implementation is typically concerned with multiple event types, type-checked data structures as illustrated previously are important. The described benefits are given independent of applied programming paradigms, be it Object-oriented Programming or Functional Programming.

The following example provides a TypeScript version of the [support ticket system example](#) from Chapter 12 ([run code](#)):

Support Ticket system: Events

```
const SupportTicketOpenedEvent = createEventType('SupportTicketOpened')
  <{ticketId: string, authorId: string, title: string, description: string}>();

const SupportTicketCommentedEvent = createEventType('SupportTicketCommented')
  <{ticketId: string, authorId: string, message: string}>();

const SupportTicketClosedEvent = createEventType('SupportTicketClosed')
  <{ticketId: string, resolution: string}>();

const SupportTicketReOpenedEvent = createEventType('SupportTicketReOpened')
  <{ticketId: string}>();
```

```
type SupportTicketOpenedEvent = InstanceType<typeof SupportTicketOpenedEvent>;
type SupportTicketCommentedEvent = InstanceType<typeof SupportTicketCommentedEvent>;
type SupportTicketClosedEvent = InstanceType<typeof SupportTicketClosedEvent>;
type SupportTicketReOpenedEvent = InstanceType<typeof SupportTicketReOpenedEvent>;

type Event = SupportTicketClosedEvent | SupportTicketCommentedEvent
| SupportTicketOpenedEvent | SupportTicketReOpenedEvent;
```

Support Ticket system: Aggregate

```
type State = {id: string, isOpen?: boolean};

const applySupportTicketEvents = (state: State = {id: ''}, events: Event[] = []) =>
  events.reduce(applyEvent, state);

const applyEvent = (state: State, event: Event): State => {
  if (event.type === SupportTicketOpenedEvent.type)
    return {...state, id: event.data.ticketId, isOpen: true};
  if (event.type === SupportTicketClosedEvent.type)
    return {...state, isOpen: false};
  if (event.type === SupportTicketReOpenedEvent.type)
    return {...state, isOpen: true};
  return {...state};
};

const openSupportTicket = ({id: ticketId, authorId, title, description}:
  {id: string, authorId: string, title: string, description: string}) => {
  if (title.length > 50) throw new Error('title length exceeded');
  if (description.length > 500) throw new Error('description length exceeded');
  return [new SupportTicketOpenedEvent({ticketId, authorId, title, description})];
};

const commentSupportTicket = (state: State,
  {authorId, message = ''}: {authorId: string, message?: string}) => {
  const ticketId = state.id;
  const events = [];
  if (!state.isOpen) events.push(new SupportTicketReOpenedEvent({ticketId}));
  events.push(new SupportTicketCommentedEvent({ticketId, authorId, message}));
  return events;
};

const closeSupportTicket = (state: State,
  {resolution}: {resolution: 'solved' | 'closed'}) =>
  [new SupportTicketClosedEvent({ticketId: state.id, resolution})];
```

Support Ticket system: Usage

```
const ticketId = generateId(), userId = generateId();

let state = {id: ''};

const openEvents = openSupportTicket({id: ticketId, authorId: userId,
  title: 'My computer is broken', description: 'There is a black screen'});
state = applySupportTicketEvents(state, openEvents);

const closeEvents = closeSupportTicket(state, {resolution: 'solved'});
state = applySupportTicketEvents(state, closeEvents);

const reopeningCommentEvents = commentSupportTicket(state,
  {authorId: userId, message: 'Hold on, it is broken again'});

const reopenedEvent = reopeningCommentEvents.find(
  event => event.type == 'SupportTicketReOpened');
console.log(reopenedEvent);
```

The event types specify their expected data structure as type arguments. Also, the code defines matching static types using the TypeScript utility `InstanceType` and introduces the combined type `Event`. The support ticket Aggregate component is mainly enriched with type annotations. For the structure of the state representation, the type `State` is defined. Overall, the runtime behavior remains largely unchanged. The runtime check in the function `commentSupportTicket()` is replaced in favor of a static type. Finally, the usage code again demonstrates the ability to differentiate between event types based on their type name. The invocation of the function `commentSupportTicket()` yields either one event or two different types. By checking for a specific type name, TypeScript can infer the correct event type and the associated data structure.



Type definition for class instances

For every class component, TypeScript implicitly defines a type with the same name that represents an instance of the class. However, when an operation dynamically defines and returns a class, TypeScript does not perform this procedure. In such cases, this mechanism can be mimicked through a manual definition with the utility `InstanceType` and the keyword `typeof`.

Dependency Inversion

The Dependency Inversion principle is typically applied through the use of interfaces. As explained in Chapter 5, JavaScript does not provide a native language construct for this concept. In contrast, TypeScript supports interfaces through the keywords `interface` and `type`. Both allow to define abstract component signatures, for which other software parts can provide specialized implementations. The implementations can optionally be marked with the keyword `implements` to enforce the contract independent of actual component use. With regard to the Domain Layer, there are two main use cases for Dependency Inversion. For one, Domain Services can be defined as interfaces, which is especially useful when there are infrastructural dependencies. Secondly, Repository signatures can be expressed as explicit Domain Model components, for which the Infrastructure part provides implementations.

The following code is a TypeScript variant of the [commenting system example](#) from Chapter 5 ([run code usage](#)):

Commenting System: Offensive Language Detector interface

```
interface OffensiveLanguageDetector {  
  doesTextContainOffensiveLanguage(text: string): boolean;  
};
```

Commenting System: Offensive Terms Detector implementation

```
class OffensiveTermsDetector implements OffensiveLanguageDetector {  
  #offensiveTerms: string[];  
  
  constructor(offensiveTerms: string[]) {  
    this.#offensiveTerms = offensiveTerms;  
  }  
  
  doesTextContainOffensiveLanguage(text: string) {  
    const lowercaseText = text.toLowerCase();  
    return this.#offensiveTerms.some(term => lowercaseText.includes(term));  
  }  
}
```

Commenting System: Comment Collection and usage

```

type Comment = {author: string, message: string};

type Options = {offensiveLanguageDetector: OffensiveLanguageDetector};

class CommentCollection {

    #comments: Comment[] = [];
    #offensiveLanguageDetector;

    constructor({offensiveLanguageDetector}: Options) {
        this.#offensiveLanguageDetector = offensiveLanguageDetector;
    }

    submitComment(author: string, message: string) {
        if (this.#offensiveLanguageDetector.doesTextContainOffensiveLanguage(message))
            throw new Error('offensive language detected, message declined');
        this.#comments.push({author, message});
    }
}

const offensiveLanguageDetector = new OffensiveTermsDetector(['stupid', 'idiot']);
const commentCollection = new CommentCollection({offensiveLanguageDetector});
commentCollection.submitComment('John', 'This is a great article!');
commentCollection.submitComment('Jane', 'You are an idiot!');
commentCollection.submitComment('Joe', 'What about KISS (Keep it simple stupid)?');

```

The interface `OffensiveLanguageDetector` defines a contract that every concrete implementation of an offensive language detection mechanism must satisfy. Its only operation `doesTextContainOffensiveLanguage()` expects a string argument and returns a boolean. The class `OffensiveTermsDetector` represents a term-based detection mechanism that implements the introduced interface. Its internal details, such as constructor arguments, are irrelevant for the abstraction. The class `CommentCollection` expects a component that is compatible with the interface `OffensiveLanguageDetector`. Upon instantiation, the passed in argument is saved as private member and used in the command `submitComment()`. Overall, the code looks similar to the original JavaScript example. However, there is one key advantage apart from the type-safety. The concept of a generic offensive language detection mechanism is explicitly defined through an interface, without referring to concrete behavior.

The second example illustrates the use of Repository interfaces for the [consultant profile](#)

directory example from Chapter 9 (run code usage):

Consultant Profile Directory: Generic Repository interface

```
interface Repository<Entity> {  
  
    save(entity: Entity): Promise<void>;  
    load(id: string): Promise<Entity | null | undefined>;  
  
}
```

Consultant Profile Directory: Consultant Profile Repository interface

```
interface ConsultantProfileRepository extends Repository<ConsultantProfile> {  
  
    findAllWithSkill(skill: string): Promise<ConsultantProfile[]>;  
  
}
```

Consultant Profile Directory: Consultant Profile in-memory Repository

```
class ConsultantProfileInMemoryRepository implements ConsultantProfileRepository {  
  
    #store = new Map<string, ConsultantProfile>();  
  
    async save(entity: ConsultantProfile) {  
        this.#store.set(entity.id, entity);  
    }  
  
    async load(id: string) {  
        return this.#store.get(id);  
    }  
  
    async findAllWithSkill(skill: string) {  
        const profiles = Array.from(this.#store.values());  
        return profiles.filter(profile => profile.skills.includes(skill));  
    }  
  
}
```

The interface Repository represents a generic definition of the basic functionality for a Repository as described in Chapter 9. Persisting an Entity is the responsibility of the

command `save()`. The operation `load()` is meant to retrieve an instance based on an identifier. Both operations are specified as asynchronous. Through the definition of a type parameter, the interface can be used for implementations with arbitrary Entity types. The interface `ConsultantProfileRepository` extends the generic Repository for the Entity type `ConsultantProfile` and defines a custom query. The operation `findAllWithSkill()` accepts a skill and yields a collection of matching consultant profiles. This signature expresses a domain-specific use case without referring to technological details. Finally, the class `ConsultantProfileInMemoryRepository` is an alternative implementation to the filesystem-based variant of the original example.

Commands and Queries

In many aspects, Commands and Queries are very similar to events. They are also messages, whose purpose is to transport specialized knowledge. Consequently, this knowledge should also have a homogenous structure across instances of the same type. As with the event type factory, the [message type factory](#) from Chapter 11 can be reworked to use static typing. One notable difference to events is that Commands and Queries are typically sent from the outside to the Application Layer. Other than local events, they are often instantiated by a client and must be de-serialized upon arrival. When using TypeScript, special care must be taken that the expected data structures are actually enforced at runtime. Many serialization operations such as `JSON.parse()` return a completely untyped value in TypeScript.

The following code provides a TypeScript implementation of the message type factory from Chapter 11 ([run code](#)):

Message: Message Type Factory

```
type IdGenerator = () => string;
type Metadata = Record<string, object>;
type MetadataProvider = () => Metadata;

let generateId: IdGenerator = () => '';
let createDefaultMetadata: MetadataProvider = () => ({});

const createMessageType = <TypeName extends string>(type: TypeName) => <Data>() => {
    return class Message {

        static readonly type: TypeName = type;
        readonly type = type;
        readonly data: Data;
```

```

readonly id = generateId();
readonly metadata: Metadata;

constructor({data, metadata = {}}: {data: Data, metadata?: Metadata}) {
  this.data = data;
  this.metadata = {...createDefaultMetadata(), ...metadata};
}

};

const setIdGenerator = (idGenerator: IdGenerator) => generateId = idGenerator;

const setDefaultMetadataProvider = (defaultMetadataProvider: MetadataProvider) =>
  createDefaultMetadata = defaultMetadataProvider;

const messageTypeFactory =
  {createMessageType, setIdGenerator, setDefaultMetadataProvider};

```

Message: Message Type Factory usage

```

const CreateUserCommand = createMessageType('CreateUser')
<{userId: string, email: string, username?: string}>();
const GetUserQuery = createMessageType(' GetUser')<{userId: string}>();

messageTypeFactory.setIdGenerator(generateId);
messageTypeFactory.setDefaultMetadataProvider(() => ({creationTime: new Date()}));

console.log(new CreateUserCommand(
  {data: {userId: '1', email: 'alex@example.com'}, metadata: {}}));
console.log(new CreateUserCommand(
  {data: {userId: '1', email: 'james@example.com', username: 'james'}}));
console.log(new GetUserQuery({data: {userId: '1'}},
  metadata: {authentication: {subjectId: '123'}}));

```

Sample Application: TypeScript implementation

This section summarizes the most important differences of a Sample Application implementation in TypeScript compared to the original JavaScript version. Overall, the Software Architecture and the program layout remain unchanged. Within the Write Model implementation

for each context, the use of static types makes selected runtime checks obsolete. For the read sides, Read Model structures are defined as explicit types, which makes them first-class citizens in the code. Within the Application Layer, generic interfaces for message-handling components enforce the naming convention of service operations. Many infrastructural functionalities, such as the Event Bus and the Event Store are equipped with type parameters. This ensures correct signatures for return values and callback arguments. Finally, the User Interface part is enriched with type annotations wherever it makes sense.



Exploring the code bundle

As mentioned in the beginning of this chapter, the code bundle contains a complete TypeScript implementation of the Sample Application. Note that this variant is not directly executable, as it misses necessary build steps. Also, the code is only one of many possible approaches of an implementation with static typing.

The discussed concepts and their examples as well as the Sample Application implementation show some possible advantages of static types. Nevertheless, this appendix does not represent a universal recommendation to use them. In fact, there are scenarios, where plain JavaScript works just as well or even better. Also, TypeScript specifically is different from a statically typed language where type information is available at runtime. In the end, the ideal choice of a programming language always depends on the specific scenario.

Appendix B: Going into production

When building software for productive use, most of the infrastructural functionalities should be powered by existing technologies. Employing self-made components is in most cases not a feasible approach. The following sections describe the main parts of an exemplary technology stack that can be used in production. This incorporates identifier generation, containerization, orchestration, an Event Bus, an Event Store, a Read Model storage and an HTTP server. The technologies are illustrated with basic examples and are partially also integrated into the Sample Application. Whenever possible, they are consumed through implementations that resemble the same interfaces as their filesystem-based counterparts. Note that the selected technologies should not to be understood as universal recommendations. The usefulness of tools always depends on the respective project and its circumstances.



The missing parts

This appendix does not cover every aspect that is necessary to build a software for production scenarios. Rather, it focuses on parts that are especially relevant for implementing DDD, CQRS and Event Sourcing. Other than in Chapter 10, the topics Authentication and Authorization are excluded. Also, the employed technologies may miss certain security-related configurations.

Identifier generation

The UUID standard version 4 is generally recommendable for identifiers, independent of specific use cases. One advantage of this standard is that there are numerous implementations across programming languages that can be used interchangeably. In theory, even the custom code shown in this book can be used in production. However, one important aspect for UUID generation is the randomness quality and the consequential likelihood of identifier collisions. The implementation introduced in Chapter 6 uses `Math.random()` and has a high chance of duplicates. Therefore, it is better to use a third-party module. Since identifiers must also be generated on the client, a library should support this use case. At the time of writing, the most popular npm package for UUID generation is [uuid](#).

The following example shows a reworked version of the function `generateId()` that uses the `uuid` npm module:

ID generation: Usage of uuid third-party module

```
import {v4 as generateId} from 'uuid';

export default generateId;
```

Containerization and Orchestration



Installation of Docker and Docker Compose

This section and the Sample Application implementation make use of Docker and Docker Compose. Executing the code examples and running the Sample Application requires to install both technologies locally. For this purpose, please refer to the [Docker installation guide](#) and the [Docker Compose installation guide](#).

Containerization and Orchestration are two important concepts for software that incorporates multiple runtime programs. **Containerization** is about packaging individual programs together with their dependencies into isolated and lightweight logical runtime units. Multiple such units can be operated on a single host without affecting each other's resources. Every container acts as its own operating system with the possibility to share network and filesystem resources with the host. One of the most popular container technologies is **Docker**. There are multiple ways to create Docker containers. For one, custom images can be built using Dockerfile manifests. This approach is especially useful for functionalities that require setup or package installation. Alternatively, pre-built Docker images can be downloaded in order to operate a container with custom code mounted into it.

The following example starts an HTTP server that responds to each request with an incrementing counter value:

Containerization: HTTP-based counter

```
let counter = 0;
require('http').createServer(
  (request, response) => response.end(`#${counter++}`).listen(50000);
```

This code can be packaged into a container with port forwarding based on the Node.js image by executing the following:

Containerization: Dockerized HTTP-based counter

```
docker run -d -p 50000:50000 -v "$PWD/counter.js":/counter.js node:alpine counter.js
```

Orchestration is the process of configuring software parts and their interaction with each other. At a basic level, it is about defining the individual units and how many instances of each one are operated. The second main functionality is to control how they can communicate with each other over a network. Typically, the configuration is written in a markup language such as YAML. Most orchestration technology is compatible with container technologies such as Docker. One of the most popular tools is Kubernetes, which is often used for software with a Microservices-based architecture. Another and more simple alternative for single-host containerized orchestration is [Docker Compose](#). In fact, this tool is often recommended for local development environments. For brevity reasons, this appendix uses Docker Compose.

The next example shows a reworked version of the HTTP-based counter using Redis as in-memory storage:

Containerization: HTTP-based counter with Redis

```
const http = require('http');
const Redis = require('ioredis');

const client = new Redis({host: 'redis'});

http.createServer(async (request, response) =>
  response.end(`${await client.incr('counter')}`)).listen(50000);
```

The last example of this section shows a Docker Compose configuration for the HTTP counter with Redis:

Containerization: Docker Compose for HTTP-based counter with Redis

```
version: '3.9'
services:
  counter:
    image: node:alpine
    ports: ['50000:50000']
    working_dir: /root
    volumes: ['./counter.js:/root/counter.js']
    command: /bin/sh -c "npm i ioredis && node counter.js"
    depends_on: ['redis']
  redis:
    image: redis:6.0.10-alpine
```

The reworked server code additionally imports the package `ioredis` and instantiates the class `Redis`. Instead of mutating a local variable, the Redis entry “counter” is incremented and returned. The Docker Compose configuration defines two parts. One is a Node.js container with port forwarding, which installs the library `ioredis` and executes the mounted script “`counter.js`”. The second container is based on the Redis image. Using the directive `depends_on`, the container startup order is customized. For executing the example, the server code must be saved as “`counter.js`” and the configuration as “`docker-compose.yml`”. The command `docker-compose up -d` starts the defined parts. Docker Compose creates a network in which containers are accessible by their name. This way, the Node.js code can connect to the Redis server via the hostname “`redis`”.



Installing third-party dependencies

Third-party dependencies such as npm modules should normally not be installed upon container creation. Instead, this step should be a part of an image build. With regard to the example, this means to create a custom Dockerfile that mounts the code and executes `npm i`. Installing dependencies only when a container is created eliminates some key advantages of containerization.

Event Store

There are many storage technologies that can be used as an Event Store. The main requirements are support for transactions and data change notifications of some kind. These two functionalities exist in a wide range of available tools. This includes many SQL databases

such as PostgreSQL as well as NoSQL stores such as MongoDB. On top of that, there are also databases that are specifically built for Event Sourcing. One such technology is [EventStoreDB](#). The stream-oriented database provides numerous specialized features, such as optimistic concurrency checks, stream subscriptions and categorized and global event streams. There are official client packages for multiple runtime environments including [Node.js](#). Compared to a generic database, EventStoreDB is more powerful for when working with event streams and requires less custom code.



Unused functionalities

The code examples in this appendix only use selected features of EventStoreDB. The two most notable unused functionalities are persistent subscriptions and projections. Persistent subscriptions enable competing consumers of a stream, where the processed revision is tracked within the database. Projections allow to create derived streams that either emit their own events or link to existing ones from other streams.

The following example shows an EventStoreDB-based Event Store component using the official Node.js client:

Event Store: EventStoreDB-based implementation

```
import {EventStoreDBClient, jsonEvent, StreamNotFoundError,
excludeSystemEvents, START, NO_STREAM, ANY} from '@eventstore/db-client';

class EventStoreDBEventStore {

    #client;

    constructor({connectionString}) {
        this.#client = EventStoreDBClient.connectionString(connectionString);
    }

    async save(streamId, events, {expectedVersion = ANY} = {}) {
        if (expectedVersion === null) expectedVersion = NO_STREAM;
        const response = await this.#client.appendToStream(
            streamId, events.map(jsonEvent), {expectedRevision: expectedVersion});
        if (!response.success) throw new StreamVersionMismatchError(
            {expectedVersion, currentVersion: response.nextExpectedRevision - 1n});
    }

    async load(streamId, {startVersion = 0} = {}) {
```

```
try {
    const fromRevision = BigInt(startVersion);
    const events = (await this.#client.readStream(streamId, {fromRevision}))
        .map(resolvedEvent => resolvedEvent.event);
    return {events, currentVersion:
        events.length > 0 ? events[events.length - 1].revision : null};
} catch (error) {
    if (!(error instanceof StreamNotFoundError)) throw error;
    return {events: [], currentVersion: null};
}
}

async subscribe(streamId, callback, {startVersion = START} = {}) {
    const callbackExecutionQueue = new AsyncQueue();
    const subscription =
        this.#client.subscribeToStream(streamId, {fromRevision: startVersion});
    subscription.on('data', ({event}) =>
        callbackExecutionQueue.enqueueOperation(() => callback(event)));
}

async subscribeToAll(callback, {startPosition = START} = {}) {
    const callbackExecutionQueue = new AsyncQueue();
    const subscription = this.#client.subscribeToAll(
        {fromPosition: startPosition, filter: excludeSystemEvents()});
    subscription.on('data', ({event}) =>
        callbackExecutionQueue.enqueueOperation(() => callback(event)));
}

}
```

The class `EventStoreDBEventStore` provides the same functionality as the filesystem-based implementation. However, it does not have the exact same interface due to specific characteristics of EventStoreDB. One difference is that stream versions are zero-based instead of one-based. This means, an empty stream does not have any version, and the first event has the revision 0. Another difference is that the global stream works with positions instead of versions. For this reason, the component implements the function `subscribeToAll()`. Also, start versions and positions for subscriptions are exclusive. This means, the first received event is the one after the specified start. One difference in data types is the use of `BigInt` for numeric parameters. The component `AsyncQueue` is utilized for a sequential execution of asynchronous subscription callbacks.

Read Model store

For the storage of Read Models, there are even more suitable technologies than for an Event Store. Consequently, it is almost impossible to make a useful universal recommendation. In general, the selected storage should have a good read performance. One decisive aspect is whether the data should be stored transiently or persistently. Depending on how the Read Model projections work, transaction support may be necessary. Another aspect is the possibility for indexing, which is required when data is accessed by multiple dimensions. Furthermore, change notifications can be useful for reactive Read Models. [Redis](#) is an easy-to-use data structure store with notification capabilities. While its stored data is by default kept in-memory, it can also be persisted. There are multiple community-driven Node.js clients, such as [ioredis](#).

The next code shows a Redis-based implementation of the indexed storage component using the third-party module [ioredis](#):

Indexed storage: Redis-based implementation

```
import Redis from 'ioredis';

class RedisIndexedStorage {

    #client; #subscribeClient; #dataType; #indexes;

    constructor({host, dataType, indexes = []}) {
        this.#client = new Redis(host);
        this.#subscribeClient = new Redis(host);
        this.#dataType = dataType;
        this.#indexes = indexes;
    }

    async update(id, updates) {
        const oldRecord = await this.#client.hgetall(`#${this.#dataType}:${id}`);
        const pipeline = this.#client.multi();
        pipeline.hmset(`#${this.#dataType}:${id}`, updates);
        const indexesToUpdate = this.#indexes.filter(index => index in updates);
        for (const index of indexesToUpdate) {
            if (index in oldRecord) pipeline.lrem(
                this.#getRedisIndexKey(index, oldRecord[index]), 0, oldRecord.id);
            pipeline.rpush(this.#getRedisIndexKey(index, updates[index]), id);
        }
        await pipeline.exec();
    }
}
```

```
}

load(id) {
    return this.#client.hgetall(`#${this.#dataType}:${id}`);
}

async findByIndex(index, indexValue) {
    const ids = await this.#client.lrange(
        this.#getRedisIndexKey(index, indexValue), 0, -1);
    return Promise.all(
        ids.map(id => this.#client.hgetall(`#${this.#dataType}:${id}`)));
}

#getRedisIndexKey(index, value) {
    return `${this.#dataType}:index:${index}:${value}`;
}

async subscribeToChanges(topic, callback) {
    const channelToSubscribe = `${this.#dataType}:${topic}`;
    await this.#subscribeClient.subscribe(channelToSubscribe);
    this.#subscribeClient.on('message', (channel, message) => {
        if (channel === channelToSubscribe) callback(JSON.parse(message));
    });
}

async publishChangeMessage(topic, message) {
    await this.#client.publish(
        `${this.#dataType}:${topic}`, JSON.stringify(message));
}

}
```

The class `RedisIndexedStorage` provides the same functionalities as the in-memory component with additional custom change notifications. Unlike the class `InMemoryIndexedStorage`, all its operations are asynchronous. As additional constructor arguments, it expects a host-name and a data type as namespace for records. The component instantiates one Redis client for read, write and publishing operations, and another one for subscriptions. Its command `update()` uses a transaction to store new data and to update affected indexes. Records are stored with their identifier as key, prefixed by the data type. Indexes follow the pattern “`<dataType>:index:<indexName>:<indexValue>`”. The command `subscribeToChanges()` subscribes to a given topic. For every message that is published on a matching channel, the callback operation is executed. Finally, the command `publishChangeMessage()` publishes a

message using the given topic.



Why custom notifications?

The custom notification mechanism of the class `RedisIndexedStorage` is meant to support the use case of reactive Read Models. Whenever a record is updated, a message can be published to inform interested subscribers about the change. As the messages go through a dedicated storage, publishers and subscribers can even be operated in different processes.

Message Bus

The main requirement for a Message Bus is a loosely coupled message exchange, where publisher and subscriber communicate only indirectly. On top of that, the distribution of Domain Events demands delivery guarantees, such as an “at least once” delivery. Typically, this requires the use of persistent queues to cope with failures and temporary unavailability. The majority of Message Bus technologies support one or more messaging protocols. One such protocol is the Advanced Message Queuing Protocol (AMQP). Another more simple variant is the Simple Text Oriented Message Protocol (STOMP). [RabbitMQ](#) is a Message Queue that supports both AMQP and STOMP as well as other protocols. The Node.js third-party library [amqplib](#) allows to communicate with RabbitMQ using AMQP. Even more, it supports specialized protocol extensions for so-called “publisher confirms”.

The first code example shows two JSON serialization operations for objects that contain `BigInt` fields:

Serialization: JSON serialization

```
const serializeJSON = input => JSON.stringify(input,
  (key, value) => typeof value === 'bigint' ? `${value}\n` : value);

const deserializeJSON = input => JSON.parse(input, (key, value) =>
  /\^\\d+n$/.test(` ${value}`) ? BigInt(value.slice(0, -1)) : value);
```

The second example provides a RabbitMQ-based implementation of the Message Bus component using the third-party module `amqplib`:

Message Bus: RabbitMQ-based implementation

```
import amqp from 'amqplib';

class RabbitMQMessageBus {

  #initialization; #subscriberGroup; #subscribersByTopic;

  constructor({connectionString, subscriberGroup = 'default'}) {
    this.#subscriberGroup = subscriberGroup;
    this.#subscribersByTopic = new Map();
    this.#initialization = amqp.connect(connectionString).then(async connection => {
      const channel = await connection.createConfirmChannel();
      channel.assertExchange('domain-events', 'direct');
      channel.assertQueue(this.#subscriberGroup, {durable: true});
      channel.consume(this.#subscriberGroup,
        message => this.#processMessage(message, channel), {noAck: false});
      return channel;
    });
  }

  async subscribe(topic, subscriber) {
    const channel = await this.#initialization;
    channel.bindQueue(this.#subscriberGroup, 'domain-events', topic);
    const newSubscribers = this.#getSubscribers(topic).concat([subscriber]);
    this.#subscribersByTopic.set(topic, newSubscribers);
  }

  unsubscribe(topic, subscriber) {
    const subscribers = this.#getSubscribers(topic);
    subscribers.splice(subscribers.indexOf(subscriber), 1);
    this.#subscribersByTopic.set(topic, subscribers);
  }

  async publish(topic, messageContent) {
    const channel = await this.#initialization;
    const json = serializeJSON(messageContent);
    return new Promise((resolve, reject) =>
      channel.publish('domain-events', topic, Buffer.from(json),
        {persistent: true}, error => error ? reject(error) : resolve()));
  }

  async #processMessage(message, channel) {
    const topic = message.fields.routingKey;
```

```
const messageContent = deserializeJSON(message.content);
await Promise.all(this.#getSubscribers(topic).map(subscriber =>
  new Promise(resolve => setTimeout(() => {
    Promise.resolve(subscriber(messageContent)).then(resolve);
  })),
));
channel.ack(message);
}

# getSubscribers(topic) { return this.#subscribersByTopic.get(topic) || []; }
```

The class `RabbitMQMessageBus` is a RabbitMQ-based alternative to the filesystem implementation. Similar to the original approach, the component works with subscriber groups. Its constructor connects to RabbitMQ and performs necessary initialization steps. First, it creates a communication channel. Afterwards, the message exchange “domain-events” and a persistent queue for the subscriber group are created if not existing. Finally, the operation `#processMessage()` is registered as callback. The command `subscribe()` ensures that the queue is bound to a given topic. The command `publish()` publishes a message under a given topic. For a guaranteed delivery, the message is made persistent and the publishing awaits confirmation. The function `#processMessage()` notifies interested subscribers and acknowledges a message. Serializing message fields with type `BigInt` is supported through the functions `serializeJSON()` and `deserializeJSON()`.

HTTP server

HTTP servers can have many individual purposes within a software. For one, Application Layer components such as Command Handlers and Query Handlers are typically exposed via an HTTP interface. Secondly, all client-side assets for a web-based User Interface must be served through a static file server. Furthermore, if a software is operated as multiple separate programs, an HTTP proxy allows to maintain a unified endpoint. The three functionalities can be powered by a single technology or a combination of many. [nginx](#) is an HTTP server that can act both as proxy and as static file server. Its behavior is controlled through a custom configuration format. Exposing Application Services via HTTP can be done using the native `http` module or third-party libraries such as [koa](#).

The following example shows an NGINX configuration for a static file server and a proxy functionality:

HTTP Server: NGINX configuration for a static file server and a proxy

```
http {
    server {
        location /static {
            root /data;
            include /etc/nginx/mime.types;
        }

        location /context-a {
            proxy_pass http://context-a;
        }

        location /context-b {
            proxy_pass http://context-b;
        }
    }

    events { }
```

NGINX configuration files consist of directives. Every directive has a name and parameters, which can be strings, numbers or blocks with nested directives. The two main parts in a configuration are the blocks `http` and `events`. While the `http` part allows use-case-specific configuration, the `events` directive defines general runtime behavior. Inside the `http` block, there can be multiple `server` entries. Similarly, inside a `server` block, there can be multiple `location` entries. Every `location` directive accepts a location match and a block. Whenever a request URL matches a location pattern, the associated behavior is applied. For the example, the `location` directive with the pattern “/static” defines the static file server functionality. The other `location` directives define a proxy behavior for the hosts “context-a” and “context-b”.

Sample Application: Going into production

This section describes an implementation of the Sample Application that employs technologies suitable for a productive use. The containerization is done via Docker and the orchestration is done via Docker Compose. For each of the three context implementations, the respective write side and read side are operated as individual containers. EventStoreDB is used as Event Store and Redis as Read Model storage. The separate contexts maintain

their own instances of both technologies. RabbitMQ is used as central Message Bus for reliable event distribution. Every context part exposes an HTTP interface using the native module `http` as illustrated in Chapter 13. Serving static assets and unifying the separate HTTP interfaces is done by a single NGINX instance. Overall, this program layout results in 14 Docker containers.

Dependencies and shared code

The use of EventStoreDB, Redis and RabbitMQ requires to introduce third-party dependencies and according infrastructural components to the existing implementation. For the dependency management, a `package.json` file is created at the root directory with entries for `@eventstore/db-client`, `ioredis` and `amqplib`. In this case, a single file is sufficient, as all context parts depend on the same technologies. The infrastructural components `EventStoreDBEventStore`, `RedisIndexedStorage` and `RabbitMQMessageBus` from the previous sections are re-used as-is. The existing source code of the Sample Application requires only minor adjustments. Both the Command Handlers of all contexts and the generic component `DomainEventPublisher` are changed to correctly work with EventStoreDB. The class `TaskBoardReadModelProjection` is modified to use the notification mechanism of Redis. Finally, the configuration file for each context part is adjusted accordingly.

As example, the following code shows the updated configuration for the write side of the task board context implementation:

Sample Application: Task Board write side configuration

```
const eventStore = new EventStoreDBEventStore(
  {connectionString: 'esdb://task-board.eventstoredb:2113?Tls=false'});

const eventBus = new EventBus(new RabbitMQMessageBus(
  {connectionString: 'amqp://rabbitmq', subscriberGroup: 'taskBoard'}));

const taskAssigneeReadModelStorage = new RedisIndexedStorage(
  {host: 'task-board.redis', dataType: 'taskAssignee', indexes: ['assigneeId']});

const commandHandlers = new TaskBoardCommandHandlers({eventStore});
new TaskBoardDomainEventHandlers(
  {eventStore, eventBus, taskAssigneeReadModelStorage}).activate();

const httpInterface = createHttpInterface(
  message => commandHandlers.handleCommand(message), ['POST']);
http.createServer(httpInterface).listen(80);
```

Containerization and Orchestration

The following example shows the relevant parts of the Docker Compose configuration for the Sample Application:

Sample Application: Docker Compose configuration

```
version: '3.9'
x-context-part: &context-part
  image: node:alpine
  working_dir: /root/code/sample-application/going-into-production
  volumes: ['../../:/root/code:ro']
x-eventstoredb: &eventstoredb
  image: eventstore/eventstore:20.10.0-buster-slim
  environment:
    - EVENTSTORE_RUN_PROJECTIONS>All
    - EVENTSTORE_INSECURE=true
    - EVENTSTORE_ENABLE_EXTERNAL_TCP=true
    - EVENTSTORE_ENABLE_ATOM_PUB_OVER_HTTP=true
services:
  project.eventstoredb:
    <<: *eventstoredb
  project.write-side:
    <<: *context-part
    command: /bin/sh -c "./await-setup.sh project && node project/write-side"
  project.redis:
    image: redis:6.0.10-alpine
  project.read-side:
    <<: *context-part
    command: /bin/sh -c "./await-setup.sh project && node project/read-side"
task-board.eventstoredb:
  <<: *eventstoredb
task-board.write-side:
  <<: *context-part
  command: /bin/sh -c "./await-setup.sh task-board && node task-board/write-side"
task-board.redis:
  image: redis:6.0.10-alpine
task-board.read-side:
  <<: *context-part
  command: /bin/sh -c "./await-setup.sh task-board && node task-board/read-side"
user.eventstoredb:
  <<: *eventstoredb
user.write-side:
  <<: *context-part
```

```
command: /bin/sh -c "./await-setup.sh user && node user/write-side"
user.redis:
  image: redis:6.0.10-alpine
user.read-side:
  <<: *context-part
  command: /bin/sh -c "./await-setup.sh user && node user/read-side"
rabbitmq:
  image: rabbitmq:3.8.11-management-alpine
  hostname: rabbitmq
nginx:
  image: nginx:1.19.6-alpine
  ports: ['50000:80']
  volumes: ['./nginx.conf:/etc/nginx/nginx.conf:ro', './:/usr/share/nginx/ui:ro']
```

The configuration first defines two templates. Entries that start with “x-“ are called extension fields and can be combined with YAML anchors to create reusable templates. Every context consists of an EventStoreDB instance, a write side, a Redis instance and a read side. The root directory of the code bundle is mounted into each Node.js container. Both RabbitMQ and NGINX are defined as single instances. For the NGINX server, the Sample Application directory is mounted and port 80 is forwarded to the host. All containers are based on pre-built Docker images. The execution commands for the context parts consist of two instructions. First, the shell script “await-setup.sh” is executed to wait for EventStoreDB, Redis and RabbitMQ. Afterwards, the actual Node.js program is started.

HTTP server configuration

The following code shows the NGINX configuration for the Sample Application:

Sample Application: NGINX configuration

```
http {
  resolver 127.0.0.11 ipv6=off;

  map $request_method $side {
    default read-side;
    POST write-side;
  }

  server {
    location ~ ^/($|[^/]*) {
      root /usr/share/nginx/ui;
```

```
index user/ui/user-login.html;
include /etc/nginx/mime.types;
}

location /project {
    proxy_pass http://project.$side;
}

location /task-board {
    proxy_set_header Connection '';
    proxy_http_version 1.1;
    chunked_transfer_encoding off;
    proxy_pass http://task-board.$side;
}

location /user {
    proxy_pass http://user.$side;
}
}

events { }
```

The directive `resolver` ensures that NGINX uses the Docker nameserver for host resolution. Otherwise, containers would not be accessible by their name. The mapped variable `$side` maps the HTTP request method to either the value “write-side” or “read-side”. Within the `server` directive, there are four nested `location` blocks. The first one defines the static file server part. For both the root path and every path with at least two components, a static asset is served. Furthermore, the file “`user/ui/user-login.html`” is used as root document. Requests with the paths “`/project`”, “`/task-board`” or “`/user`” are forwarded to the respective context. POST requests are forwarded to the write side of a context, all others to the read side. The task board part contains additional configuration for SSE support.

Running the application

The Sample Application can be started by executing `npm start` within the bundle directory “`sample-application/going-into-production`”. For simplicity reasons, all dependencies are installed on the host and mounted into the containers. Stopping the software is done via executing `npm stop`. Port 80 from the NGINX container is forwarded to port 50000 on the host. This means, the Sample Application is accessible at <http://localhost:50000/>. For debugging

purposes, the Docker Compose configuration also exposes ports from all EventStoreDB and Redis instances as well as the RabbitMQ container. The ports are forwarded to the local port range 50001-50007. Both EventStore and RabbitMQ provide a web-based User Interface. For RabbitMQ, the required username and password is “guest”. Redis does not provide a UI, but can be accessed via a client.

The reworked Sample Application demonstrates how to operate a software based on DDD, CQRS and Event Sourcing in production. One additional step would be to scale individual context parts beyond a single instance. For this purpose, the NGINX server must be adjusted to also act as load balancer. The write side of each context can be scaled without code adjustments. For the read sides, the Read Model projections must either act as competing consumers or must be sharded.

Bibliography

- Evans, Eric. 2004. Domain-Driven Design: Tackling the Complexity in the Heart of Software.
- Evans, Eric. 2012. Domain-Driven Design Reference.
- Vernon, Vaughn. 2013. Implementing Domain-Driven Design.
- Young, Greg. 2014. CQRS and Event Sourcing - Code on the Beach 2014. [Talk on YouTube](#)
- Young, Greg. 2014. GOTO 2014 â€¢ Event Sourcing. [Talk on YouTube](#)
- Young, Greg. 2016. A Decade of DDD, CQRS, Event Sourcing. [Talk on YouTube](#)
- Fowler, Martin. 2017. What do you mean by “Event-Driven”?.. [Web page](#)