wolkenkit

The semantic JavaScript backend
for event-driven development

made with Love by the native web.

wolkenkit

The semantic JavaScript backend for event-driven development

Written by

Golo Roden, Matthias Wagler, Susanna Roden

Published by

the native web GmbH
Hauptstraße 8
79359 Riegel am Kaiserstuhl
Germany

Part I: Why **wolkenkit**

Software development is not an end in itself. Instead, software gets written to solve actual real-world problems. Unfortunately, this fails regularly for various reasons. That‘s why we have built wolkenkit, a semantic JavaScript backend that addresses three important aspects of software development, and that empowers you to build better software faster.

Empowering interdisciplinary teams

While trying to build software that solves real-world problems, more often than not issues arise. Important questions on the underlying business problems can‘t be answered, as the technical experts are missing the required domain knowledge. Since talking to the domain experts can feel intimidating, it becomes hard to answer these questions. Nevertheless this is the only viable way to grasp and understand the details of the domain.

Unfortunately, talking to people from other domains is rarely encouraged in education, and you are used to building teams and even organisations as clusters of similarly qualified people. As a result birds of a feather flock together.

We believe that having interdisciplinary teams with open discussions dramatically improves software development. But even within an interdisciplinary team one of the hardest things is finding consensus on what the actual underlying problem is that needs to be solved. What is the core domain? Which problem is the user going to solve, and how shall they do that?

It is incredibly hard to find answers to these questions. As developers we are so used to thinking in terms of *CRUD*, that *create*, *update*, and *delete* are the only verbs we know to map reality to. Yet the truth is that every non-trivial real-world problem is way more complex than those three words, and requires more expressive powers.

Domain-driven design
wolkenkit uses your domain language as code. This way it invites you to work in an interdisciplinary team as early as possible.

With wolkenkit software development is different, as it builds on the principles of *domain-driven design*. Before writing any code, model, discuss, and shape your core domain on a whiteboard, together with the people that know it inside out. Then transform the result into Java-Script code, and run it with wolkenkit. We have carefully designed and developed wolkenkit to make this transformation as frictionless as possible.

Learning from your past

For several decades developers have been getting used to store the current application state. Once you update it, any previous information gets lost. This leads to a number of questions that can‘t be answered easily. What was the previous state? What was the intention of updating? How did state evolve over time?

There are several workarounds for this problem, including history tables and audit logs. However, these solutions do not address the root cause of the problem, as they follow a data-driven approach, not a domain-driven one. They do not capture the users‘ intentions, only their results. Compare this to memorizing a fact, but immediately forgetting the events and experiences that have brought you there.

Event-sourcing
wolkenkit does not store the current application state, but the stream of events that led to it. This allows you to reinterpret your past.

wolkenkit uses a different approach, as it builds upon the principles of *event-sourcing*. It uses an *event store* to record all of the events that happen within your application. The current state is the result of this event stream. Additionally, you can replay this stream to reinterpret events and learn from your past. We have carefully designed and developed wolkenkit to make this as simple and performant as possible.

Scaling with confidence

The cloud‘s biggest benefit is its ability to scale elastically, according to your needs. Unfortunately, your application does not automatically benefit from this. Instead, your application's architecture needs to support this. This is hard to get right when building a CRUD monolith.

The biggest issue is the missing ability to optimize a CRUD application for reading and writing individually at the same time. You have to favor one side and either use a normalized or a denormalized model. The normalized model is great for consistent writes because there is no duplicated data. On the other hand it is hard to read since you need to rejoin the data. The denormalized model can be read efficiently, but it is hard to keep things consistent.

CQRS
wolkenkit separates reading data from writing them. This way you can optimize and scale them as needed.

With wolkenkit scalability is different, as it builds upon the principles of *CQRS*. It separates reading data from writing them, so you can use an optimized model for either side. This way you can combine consistent writes with efficient reads. Since wolkenkit is a distributed application, it even runs dedicated processes for reading and writing. Anyway, this separation requires some synchronization. We have carefully designed and developed wolkenkit so that you do not have to care about this synchronization, and instead are able to focus on your domain.

Part II: Principles of **wolkenkit**

With wolkenkit software development is different, as it builds upon the principles of *domain-driven design, event-sourcing* and *CQRS*. But what are these principles about? How do they engage with each other? We will have a closer look at each of those principles to give you a solid and reliable basis for your further experiences with wolkenkit.

Event-driven architecture

There may be a vast number of interpretations what *event-driven architecture* actually means, and when we are talking about it, we usually think about three things at once: The *command and query separation pattern* (CQS), *event-sourcing* and *domain-driven design* (DDD).

These three are the building blocks of event-driven architecture and before getting into detail we need to clarify what these three terms are all about and how they relate to each other.

Command and query separation

The CQS pattern describes functions to be either commands or queries. While commands affect state, queries don‘t. On the other hand, while commands don‘t return anything, queries do. So one is for writing state, the other one for reading:

```
// Command: Write state
user.login({
  login: 'user',
  password: 'secret'
});

// Query: Read state
const isUserLoggedIn = user.isLoggedIn();
```

What‘s especially important is that a given function is always either a command or a query, but never both at the same time. In other words: A function that changes state must always be of return type *void*, a function that returns data must never do any changes.

Event-sourcing

Event-sourcing describes a technique to track state changes over time: Instead of actually changing state you always only append changes, like in a book of records. The current state can then be retrieved by replaying all of the state changes.

In the following example, to get the current text of the message you need to replay all the events that relate to the `text` field, in the right order:

Order	Event	Data
1	sent	`{ text: 'hey', likes: 0 }`
2	edited	`{ text: 'Hey' }`
3	edited	`{ text: 'Hey ;-)' }`

So, in the end, you know that the message‘s current text is `Hey ;-)`. But event-sourcing allows you to do even more: As you keep track of the semantics you can easily run reports on the book of records. Questions you may want to answer include:

- Has the text ever been edited?
- How often was the text edited?
- Was the message different every time it was edited?
- …?

If you also save the point in time when events actually happen, you will be able to reset your application‘s state to any particular point in time. Additionally you gain the opportunity to put events into a temporal relation to each other. This allows you to answer questions such as:

- What is the typical duration between two edits?
- How long does it take until a message is edited for the first time?
- …?

All these questions can be answered easily, without even having known about them in advance. The secret sauce is not to store a simple *UPDATE* in a *CRUD* data-base, but to store semantics, i.e. the intention of the user that caused the *UPDATE*.

The things that you store are called *events* because they have happened and cannot be undone. They are facts that describe the truth.

All the events are stored in an append-only data store, the so-called *event store*. As an event store only needs to support *INSERT* and *SELECT* and is basically nothing but a simple table, basically any arbitrary data store is a good match.

Although they may differ in performance, conceptually it doesn‘t matter if you use a SQL or a NoSQL database, or no database at all. Even using plain text files with a decent file system is fine.

Domain-driven design

When talking about event-driven development, DDD is often described last and is way harder to grasp than CQS and event-sourcing, but at the same time it‘s unfortunately also the most important one because it sets the foundation for creating meaningful commands and events. So make sure that you get this right.

The basic idea of DDD is about formalizing language:

As context is important in language, the same words may mean different things in different contexts. So the primary goal is to identify these contexts and set up a common language that is used by anyone within this context. This is called the *ubiquitous language*.

No universal language
The ubiquitous language is context-dependent, and hence not universal. It is perfectly fine for the same word to have different meanings in different contexts.

When modeling a domain, hence you do not try to create *one* language for the whole domain, but you divide the domain into multiple parts where each part has a ubiquitous language.

These parts are so-called *bounded contexts*. A bounded context must not necessarily be visible in code, it‘s just a boundary for the ubiquitous language.

Bounded contexts differ from sub-domains. While bounded contexts are part of the *solution space*, sub-domains are part of the *problem space*. Hence they may map to each other perfectly, but in reality most often they won't: Instead, a bounded context may span multiple sub-domains.

Supposed you want to model a company using domain-driven design: While the company itself is clearly the domain, its departments are sub-domains. Anyway, when departments interact with each other, there may be a relationship with a shared language. Hence, these departments make up a single bounded context, at least partially. So, the departments are an artefact of reality (i.e., the *problem space*), but the bounded context is an artefact of the domain model (i.e., the *solution space*).

Inside of bounded contexts there are *aggregates*. They are objects that ensure the validity of invariants, and are sometimes referred to as *transactional boundaries*. Their primary task is to run business processes and to make decisions regarding them. Aggregates handle commands and publish events.

E.g., when a user wants to edit a message they send an `edit` comand with the new text and the ID of the message‘s aggregate.

```
const command = new Command({
  context: {
    name: 'communication'
  },
  aggregate: {
    name: 'message',
    id: 'ec211ad3-484c-417b-99e4-391fe9983b12'
  },
  name: 'edit',
  payload: {
    text: 'Hey ;-)'
  }
});
```

The aggregate then decides whether the user is allowed to edit the text in the way they want to. There may be some restrictions, e.g. there may be the rule that the text must not be empty.

If the aggregate allows to run the `edit` command, it creates an `edited` event and publishes that. Now the wish has been transformed to a fact: While the aggregate could reject the wish, the fact is now evident and cannot be undone. Even if the business rules change in the future this will only affect future decisions. The decisions made before stay untouched as a relict of the past.

Expressive verbs, please
Avoid *create*, *update*, and *delete* as verbs. There is almost always a better word to express what you mean. *Lists* and *collections* often point out to missed domain concepts.

CQRS = CQS in the large

Now, basically *CQRS* (command query responsibility seggregation) is nothing but CQS applied on the system‘s architecture level, combined (most often) with event-sourcing and DDD. What this essentially means is that a system is split into a *write* and a *read part*, according to CQS: While the write part handles *commands*, the read part handles *queries*.

This is because in modern web applications there are more requests that want to read data, than there are requests that want to write data. Hence scalability should favor readability. Also, you have to consider the *CAP theorem*.

CAP and eventual consistency

The CAP theorem implies that any distributed application can only fulfill two of the three following requirements at the same time: Having *consistent* data across all nodes at any point in time (C), being always *available* (A), and being tolerant to *network partitions* (P).

Counterintuitively dropping consistency does not hurt as bad as you‘d expect it.

What you end up with is the so-called *eventual consistency.* This means that your application may show stale data, but eventually it will be consistent across all nodes. Of course you have to carefully think about the implications of eventual consistency and decide whether it‘s a viable way. In the end, it depends on the context.

This results in effects you know from large distributed web applications, such as Twitter or Facebook: When you post something in your timeline or tweet something, the message shows up. If you reload, it may or it may not appear. If it doesn‘t appear, it will after a few seconds. This is an evidence of eventual consistency, as not all nodes have been updated at the same time: Depending on which node you accidentally get connected to, it will already have the data or not.

Nevertheless consistency is far less important than assumed: The usual example where consistency is required is a financial transaction.

In fact, not even credit institutes work with transactions, they too rely on eventually being consistent.

Dealing with (business) risk

Otherwise, if you transfer money to another account and mistype its number, the transaction would fail immediately. But it doesn‘t. Instead, the credit institute accepts your order (which, actually is a command), and tries to fulfill it. Once they realize that they cannot fulfill your order, they create a compensating transaction.

What‘s especially important is that they do not rollback your order. In fact, they can‘t, as (technically) there is no transaction.

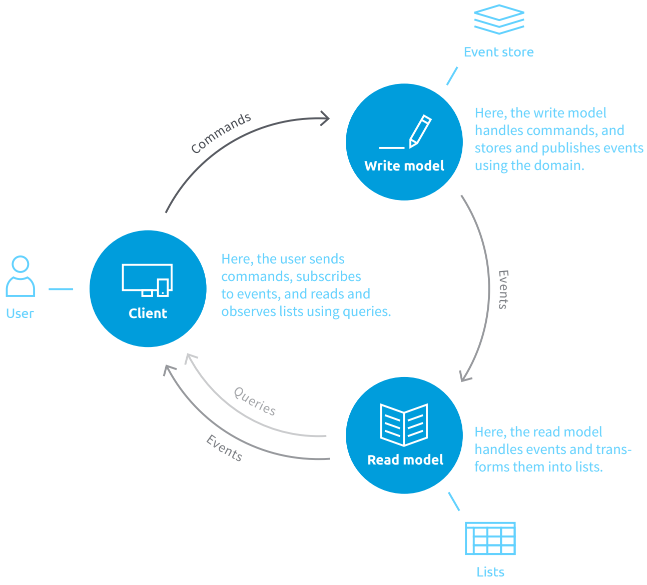
Credit institutes have perfected the art of dealing with this, as they treat all things that may go wrong in a distributed system as business risks, and most often they have found a way to turn them into profit centers.

Supposed you want to get money from an ATM, but the ATM is currently disconnected and hence can not verify whether it is fine to debit your account or not. So what do they do?

Of course, they give you the money. First, they can charge interest if you overdraw your account, so they even make more money. Second, giving money to someone poor although you shouldn‘t makes up a much better headline than refusing money to a billionaire. And by the way, nobody complains about having some additional money.

Anyway, there are situations that eventual consistency is a bad idea for. Think of everything with high risk, e.g. when your application deals with life, such as medical applications. If you need to control lasering eyes, an aircraft, or a nuclear power station, you probably don‘t want to be *eventually* consistent. Anyway, most web applications are not comparable to this.

The structure of a CQRS system


Event store
Commands
Write model
Here, the write model handles commands, and stores and publishes events using the domain.
User
Client
Here, the user sends commands, subscribes to events, and reads and observes lists using queries.
Events
Queries
Events
Read model
Here, the read model handles events and transforms them into lists.
Lists

In a CQRS system, you typically have a client with a task-based UI. This may be a static web site, a mobile application, or anything else. Everytime the user performs a task, the client sends one or more commands to the server.

The server acknowledges the commands‘ receipt. For now, the client is done. This means that the client does not know whether the command was handled successfully in the first place. Instead, it‘s fire-and-forget. The reason the client does not wait for a response is that the UI shall not block, and handling the command may take a while.

Simulate transactions

To avoid waiting indicators the UI may simulate results until they are actually available. This improves usability, but requires errors to be handled asynchronously.

The server typically stores the command in a queue to decouple receiving commands from handling them. The queue allows the workers behind the queue to scale independently, as it acts as a buffer.

These workers are called *command handlers*. They decide which aggregate the command refers to, load and run the appropriate domain logic on it. This is where event-sourcing and domain-driven design come into play. We‘ll look at this in a minute.

The aggregate then decides whether to run the command. It may be that a command is not allowed in a certain state of the aggregate, e.g. the text of a message can only be edited if the message is not older than 24 hours. Anyway, the aggregate publishes events on what has happened. This way, you always get an event, even in case of an error.

Finally, the command handler forwards those events to anyone who is interested. This is done by storing them to a queue again.

Wishes versus facts

Commands are *wishes*, events are *facts*. While commands can be rejected, events can‘t be undone: They *have* happened.
Hence, commands use the imperative (**`edit`**), events use past tense (**`edited`**).

To display results the client could subscribe and react to events. This enables live-updates, but makes it incredibly hard to get the initial view. For that, the events must be materialized into a snapshot the client can fetch at any point in time.

This is done using *event denormalizers*. They again are workers, but handle events instead of commands. In their most essential form they map events to CRUD. Each denormalizer is responsible for a table that backs a specific view in the client and updates it accordingly.

This way, each time an event is received by a denormalizer, the view gets updated.

As you can easily run more than one instance of a denormalizer, scaling them is very easy.

Equal, but maybe delayed
Since the denormalizers aren‘t synchronized, their respective tables may not be updated simultaneously, but delayed. This is what *eventual consistency* is all about.

So, the client fetches these tables. As there is a dedicated table for each of the client‘s views, a simple `SELECT * FROM table` is usually enough. This works because the denormalizers do not care about 3rd normal form but store data in a denormalized form perfectly suited for each view. Additionally, clients may subscribe to the events themselves, e.g. to display updates in real-time.

In summary, clients send commands to the domain which, as a reaction, publishes appropriate events. These events are then used to make up materialized views. All the events are stored within an event store.

Some thoughts on the event store

The event store is one of the central concepts in a CQRS architecture. But, as any read requests go to the materialized views, the event store is primarily used for writing.

Yet, there may be a performance bottleneck: The more events are stored, the longer it takes the event stream to load. As the event store is append-only, every new event makes reading and applying the event stream more complex and time-consuming.

Fortunately, there is a way to mitigate this: Since applying the very same events always results in the same state, it may be cached easily. Such a cached state is called a *snapshot*, and it can be taken automatically by the event store in the background.

To save disk space you may delete any events up to the latest snapshot, but this limits your options to analyze data.

Having an event store also provides another benefit. As it is the *single source of truth* for the entire system, it is the only component that must be backed up regularly. Anything else can be rebuilt from the event store.

This is especially true for the materialized views that were created by the denormalizers. Supposed one of them goes down, it can be easily restored by *replaying* the event stream from the event store, and start again to receive and handle events.

Replaying not only helps to recover crashed denormalizers. The very same mechanism also helps you to deal with short outeages, or even with bringing up completely new denormalizers that you didn‘t have before.

This is useful if you want to run an analysis nobody thought about before. All you need to do is to setup a denormalizer that reacts to the events important for the analysis, processes the data in the desired manner, and stores them in a materialized view.

Process managers

So far, we assumed that every command only affects a *single* aggregate. What if it doesn‘t? What if a command spans *multiple* aggregates, possibly stretched over time? The most obvious solution would be to write a command handler that updates two aggregates. Unfortunately, this is the wrong way to do it in CQRS.

The problem is that aggregates, as stated earlier, *ensure the validity of invariants, and are sometimes referred to as transactional boundaries.* In fact, in CQRS you have an all-or-nothing approach with respect to ACID for a single aggregate. This means that you cannot update multiple aggregates within one transaction.

Fortunately, there is a concept called *process manager* that deals with this. Sometimes process managers are also referred to as *sagas*. Effectively, they both mean the same, and hence are interchangeable.

A process manager is a long-running workflow that spans multiple aggregates and guarantees that all of them are updated. In other words, a process manager acts as a replacement for distributed transactions.

The way they work is quite easy: Technically, a process manager is a finite state machine that reacts to events and publishes commands. This way process managers act mirror-inverted, as they swap the roles of commands and events.

Flows
In wolkenkit, process managers are called *flows*. There are *stateful* and *stateless* flows.

By using persistence, process managers are even able to support very long-running transactions. In contrast, for simple scenarios they may also omit their state completely and act as simple *if-this-then-that*-rules. To link commands and events there are a couple of IDs, such as the *correlation ID* and the *causation ID*.

Summary

An event-driven architecture using CQRS, event-sourcing, and DDD has various advantages.

It puts interdisciplinary communication at the center of your development process. Thus you are able to create applications that reflect your domain language and help everybody to talk about actual domain problems.

Storing the run of events instead of capturing the status quo using the built-in time machine lets you parse and analyse the stream of events from your past, and encourages learning by re-interpreting these historical data.

Finally, an event-driven architecture such as wolkenkit grows with your needs: Use its cloud-ready, containerized, and distributed architecture to gain infinite horizontal scalability and simple deployments, even if you run it in your private data center.

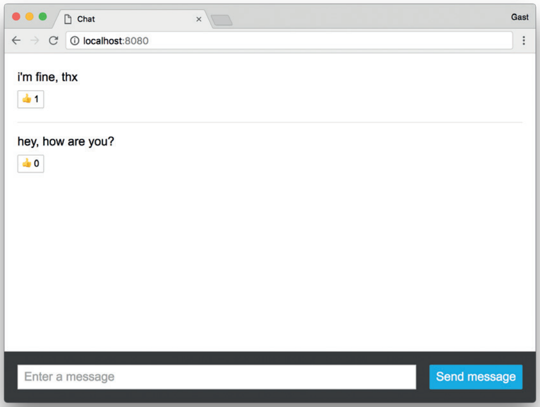
Part III: Using **wolkenkit**

Now that you know the principles of wolkenkit it‘s time to build your first event-driven application. We have prepared a guide that teaches you to create an application from scratch.

To follow this guide you need to install wolkenkit to your machine. Find the details at **https://docs.wolkenkit.io**, *Getting started* > *Installing wolkenkit*.

Setting the objective

Your application will be a chat that allows sending and receiving messages in order to talk to other users. Additionally, users are able to like messages:



i'm fine, thx

👍 1

hey, how are you?

👍 0

For this, you will implement the following features:

- Users can send messages
- Users can like messages

- Sent messages are visible to all users
- When a user enters the chat they are shown the previously sent messages
- When a user receives a message the UI is updated in real-time
- Sending and receiving messages is possible using an API
- Sending and receiving messages is encrypted

To avoid that developing your first application goes beyond the scope of this guide, there are a few constraints that limit its scope:

- There are no chat rooms
- There are no private messages
- There is no authentication
- There is no authorization

Now, let‘s get started and do some modeling with your team!

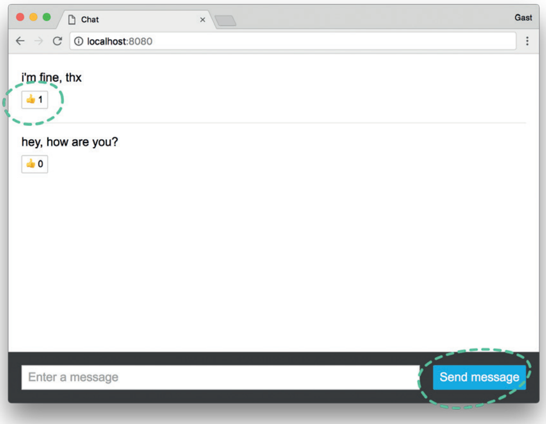
Modeling with your team

You are probably tempted to start thinking about the technical parts of the application, such as encrypting messages, having an API, and so on.

Anyway, this is not the application‘s core domain. You don‘t create the application because you want to do encryption or build an API. You create the application because you want to enable people to talk to each other by sending and receiving messages. So, for now, let‘s focus on the actual domain.

Take your time to discuss
As development is a team process, you will discuss a lot to find the right words. Take your time, play with variations, and do not stop before you have reached consensus.

When modeling an application, the first thing to do is to identify the user actions that update the application state. For that have a look at the following image:



As you can see, there are a text box with a *Send message* button, and below each message a button to like it. These actions are the *commands*. As terms we choose *send message* and *like message*.

It may be obvious to name the commands *send message* and *like message*, but there would have been alternatives. E.g., instead of *like* you might also have used *react*, *up-vote*, or *mark*. Each of these words comes with a different meaning, and you have to look for the word that best matches the intention of your core domain.

Maybe you have come up with something that uses words such as *create* and *update*. Keep in mind that this usually is not the language of your users to describe their needs. Developers often tend to use technical terminology such as *create*, *read*, *update*, and *delete*. Avoid these words when discussing the domain language, as these words are very generic and do not carry much semantics. What you get from this is a better understanding of your core domain.

Commands will write
As commands are only about writing to the application, there is no *receive message* command. Receiving messages means reading from the application.

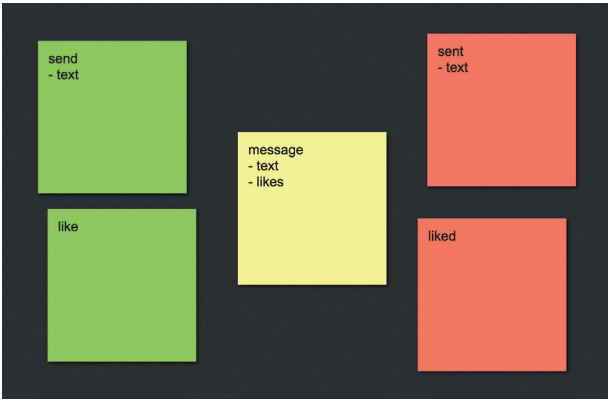
Next we have to come up with the names of the *events*.

An event describes a fact that has happened and can not be undone. Hence they are phrased using *past tense*. This way we get the two events *sent message* and *liked message*.

When the user runs a command, there must be something that handles it and decides whether it is allowed to update the application state. For that, you need an *aggregate* that embraces commands and events, and that contains the state it needs to make decisions.

In our simple example there are now two options. You could model the entire chat as a single aggregate, or have an aggregate that represents a message. We decide to use an aggregate named *message*, since there are no domain constraints regarding the chat itself. Additionally, every message needs its individual state, e.g. to store the number of likes it has received.

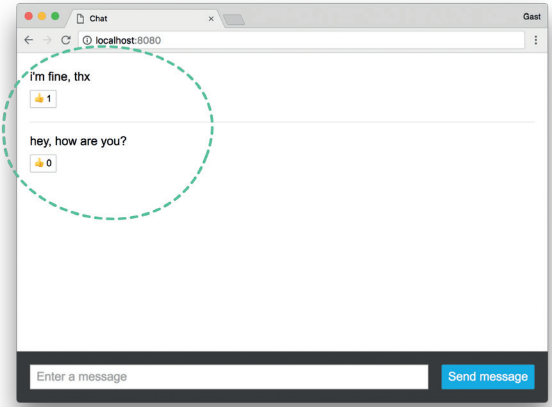
Since the aggregate is now called *message*, there is no need to repeat the aggregate‘s name inside of every command and event. So, *send message*, *like message*, *sent message*, and *liked message* simply become *send*, *like*, *sent*, and *liked*. Now, your whiteboard might look similar to this:



In more complex applications you will have multiple aggregates. While some of them will be closely related to each other, others will address completely different parts of your application. To group related aggregates use a *context*. Although we only have a single aggregate, we need to define a context. Let‘s call it *communication*.

That‘s it for the write model.

If you now have another look at the image below, you will realize that the list of messages is still missing, which is another important part of the application:



i'm fine, thx

👍 1

hey, how are you?

👍 0

In contrast to aggregates, commands, and events, rea-ding a list does not update the application state. Hence defining a list does not belong to the write model, but to the read model.

So, we define a list called *messages* with the following structure:

id	**timestamp**	**text**	**likes**
…	1484145874599	hey, how are you?	0
…	1484145883548	i‘m fine, thx	1

Now, let‘s start to code by creating the write model!

Creating the write model

First, you need to create a new directory for your application. Call it `chat`:

```
$ mkdir chat
```

Inside of this directory you will store the wolkenkit application as well as any related files, such as documentation, images, and so on. The actual wolkenkit code must be in a directory called `server`, so you need to create it as well:

```
$ mkdir chat/server
```

Finally, for the write model, you need to create another directory called `writeModel` inside of the `server` directory:

```
$ mkdir chat/server/writeModel
```

To have a valid directory structure, you also need to add three more directories that you are going to need later, `readModel`, `readModel/lists`, and `flows`:

```
$ mkdir chat/server/readModel
$ mkdir chat/server/readModel/lists
$ mkdir chat/server/flows
```

One thing every wolkenkit application needs is a `package.json` file within the application‘s directory. This file contains some configuration options that wolkenkit needs to start the application.

Create a `package.json` file in the `chat` directory:

```
$ touch chat/package.json
```

Then, open the file and add the following code:

```
{
  "name": "chat",
  "version": "0.0.0",
  "wolkenkit": {
    "application": "chat",
    "runtime": {
      "version": "1.0.0"
    },
    "environments": {
      "default": {
        "api": {
          "address": {
            "host": "local.wolkenkit.io",
            "port": 3000
          },
          "allowAccessFrom": "*"
        },
        "node": {
          "environment": "development"
        }
      }
    }
  }
}
```

To create the *communication* context, create an appropriate directory within the `writeModel` directory:

```
$ mkdir chat/server/writeModel/communication
```

To create the *message* aggregate, add a `message.js` file to the `communication` directory:

```
$ touch chat/…/communication/message.js
```

Then, open the file and add the following base structure:

```
'use strict';

const initialState = {
  isAuthorized: {
    commands: {},
    events: {}
  }
};

const commands = {};
const events = {};

module.exports = {
  initialState, commands, events
};
```

As you have learned while modeling, a message has a `text` and a number of `likes`. For a new message it makes sense to initialize those values to an empty string, and the number `0` respectively. So, add the following two properties to the `initialState`:

```
const initialState = {
  text: '',
  likes: 0,
  // ...
};
```

Now let‘s create the *send* command by adding a `send` function to the `commands` object. It receives three parameters, the `message` itself, the actual `command`, and a `mark` object.

Inside of this function you need to figure out whether the command is valid, and if so, publish an event. Afterwards you need to mark the command as handled by calling the `mark.asDone` function. In the simplest case your code looks like this:

```
const commands = {
  send (message, command, mark) {
    message.events.publish('sent', {
      text: command.data.text
    });

    mark.asDone();
  }
};
```

Please note that you need to add the text that is contained within the command to the event, because the event is responsible for updating the state.

Additionally, this gets sent to the read model and to the client, and they both are probably interested in the message‘s text.

Although this is going to work, it has one major drawback. The code also publishes the `sent` event for empty messages, as there is no validation. To add this, check the command‘s `data` property and reject the command if the text is missing:

```
send (message, command, mark) {
  if (!command.data.text) {
    return mark.asRejected('Text is missing.');
  }

  // ...
}
```

To make things work, you also need to implement a handler that reacts to the *sent* event and updates the aggregate‘s state. For that add a `sent` function to the `events` object. It receives two parameters, the `message` itself, and the actual `event`.

Inside of this function you need to update the state of the message. To set the text to its new value, use the `setState` function of the message object:

```
const events = {
  sent (message, event) {
    message.setState({
      text: event.data.text
    });
  }
};
```

Implementing the *like* command is basically the same as implementing the *send* command. There is one exception, because *like* is self-sufficient and has no additional data. Hence the `like` command could look like this:

```
const commands = {
  // ...
  like (message, command, mark) {
    message.events.publish('liked');

    mark.asDone();
  }
};
```

Anyway, this raises the question how an event handler should figure out the new number of likes. This is especially true for a client that does not have the current state at hand, but might also be interested in the *liked* event. To fix this, calculate the new number of likes and add this information when publishing the *liked* event:

```
// ...
message.events.publish('liked', {
  likes: message.state.likes + 1
});
// ...
```

Implementing the *liked* event is exactly the same as implementing the *sent* event. Hence, your code looks like this:

```
const events = {
  // ...
  liked (message, event) {
    message.setState({
      likes: event.data.likes
    });
  }
};
```

By default, you will not be able to run the new commands or receive any events for security reasons. As we are not going to implement authentication for this application, you need to allow access for public users.

For that, add the following lines to the `isAuthorized` section of the `initialState`:

```
const initialState = {
  // ...
  isAuthorized: {
    commands: {
      send: { forPublic: true },
      like: { forPublic: true }
    },
    events: {
      sent: { forPublic: true },
      liked: { forPublic: true }
    }
  }
};
```

Yay, you have created your first write model! For the client, we are now missing the list of messages, so let‘s go ahead and start creating the read model!

Creating the read model

To create the *messages* list, create a `messages.js` file within the `lists` directory:

```
$ touch chat/server/readModel/lists/messages.js
```

Where is the context?

The read model is able to handle events from multiple contexts, so it may not be possible to assign a read model to a specific context.

Then, open the file and add the following base structure:

```
'use strict';

const fields = {};
const when = {};

module.exports = { fields, when };
```

As you have decided while modeling that each message in the list of messages should have a text, a number of likes, and a timestamp, you need to define the appropriate fields:

```
const fields = {
  text: { initialState: '' },
  likes: { initialState: 0 },
  timestamp: { initialState: 0 }
};
```

Additionally, an `id` field is created automatically.

The next question is how the list becomes filled with messages. For that you need to handle the events that have been published by the write model.

Whenever a message has been sent, add it to the list of messages, and set the text and timestamp to the data that are provided by the event.

Add a `communication.message.sent` function to the `when` object. It receives three parameters, the `messages` list itself, the actual `event`, and a `mark` object.

Add the message to the list by calling its `add` function. You do not need to set the `id` field, as it gets automatically populated using the aggregate's id that is given in the event. Afterwards you need to mark the event as handled by calling the `mark.asDone` function:

```
const when = {
  'communication.message.sent' (
    messages, event, mark
  ) {
    messages.add({
      text: event.data.text,
      timestamp: event.metadata.timestamp
    });

    mark.asDone();
  }
};
```

Handling the *liked* event is similar to the *sent* event. The only difference is that you need to update an existing message instead of adding a new one. So, add an event handler, but this time call the list's `update` function:

```
const when = {
  // ...
  'communication.message.liked' (
    messages, event, mark
  ) {
    messages.update({
      where: { id: event.aggregate.id },
      set: { likes: event.data.likes }
    });
    mark.asDone();
  }
};
```

Yay, congratulations! You have created your first read model, and clients can read and observe it in real-time! Now we are ready for creating the client, so let's go ahead!

Creating the client

As wolkenkit is a backend framework, you are completely free to create whatever client you want to, as long as it is able to do HTTP requests.

For JavaScript, there is a client SDK that internally uses this HTTP API, but adds a convenience layer that simplifies talking to your backend dramatically. You can use it inside the browser as well as on the server.

Typically, you will add a `client` directory to your application's directory, but we have already prepared this for you. Just follow along without creating the directory manually.

To make things easy we have prepared a sample client for you that you are going to extend.

Download the client from **https://docs.wolkenkit.io**, *Guides > Creating an application from scratch > Creating the client* to your `chat` directory and, from within this directory, run the following commands:

```
$ tar -xvzf client.tar.gz
$ rm client.tar.gz
```

Vanilla JavaScript
The client does not depend on a specific UI framework, so you do not need any special knowledge besides what you know about vanilla JavaScript anyway.

First, you need to reference the client SDK from within the `index.html` file. For that, open the file and add the following line:

```
<script
  src="/lib/wolkenkit-client.browser.min.js"
>
</script>
```

Then, you need to connect to the backend. For this, open the `index.js` file and add the following lines:

```
wolkenkit.connect({
  host: 'local.wolkenkit.io',
  port: 3000
}).
  then(chat => {
    // ...
  }).
  catch(err => {
    console.error(err);
  });
```

To send a message, you must add an event handler to the `submit` event of the client's send message form. Inside of this handler, you can then run the `send` command of the `message` aggregate, that you can access using the `communication` context of the `chat` application:

```
document.querySelector('.send-message-form').
  addEventListener('submit', event => {
    event.preventDefault();

    const text = document.
      querySelector('.new-message').value;

    chat.communication.message().send({
      text
    });
  });
```

To get notified when something goes wrong, add the `failed` callback to the command. Also, it might be useful to reset and focus the text box, once the command has been delivered to the server. For that, add the `delivered` callback to the command:

```
chat.communication.message().send({ text }).
  failed(err => console.error(err)).
  delivered(() => {
    document.
      querySelector('.new-message').value = '';
    document.
      querySelector('.new-message').focus();
  });
```

To ensure that the text box is automatically focused when the client is opened, add another line in the end:

```
document.querySelector('.send-message-form').
  addEventListener('submit', event => {
    // ...
  });

document.querySelector('.new-message').focus();
```

Although you are now able to send messages, your client will not receive any of them. To make things work, you need to read and observe the `messages` list and update the UI accordingly. For that, use the `started` and the `updated` callbacks. As before, you will also want to make sure that you get notified in case of errors:

```
// ...
document.querySelector('.new-message').focus();
chat.lists.messages.readAndObserve().
  failed(err => console.error(err)).
  started(render).
  updated(render);
```

In a chat it makes sense to have the newest messages at the top of the client, so we will order the messages reversed by their timestamp. Also, you probably do not want to receive all messages that have ever been written, so let's limit their number to `50`:

```
chat.lists.messages.readAndObserve({
  orderBy: { timestamp: 'descending' },
  take: 50
}).
  failed(err => console.error(err)).
  started(render).
  updated(render);
```

You are now able to send and receive messages, so you already have a working chat.

What is render?

The `render` function does not belong to wolkenkit. Instead it is a ready-made function of the client blueprint that makes it easy to update the UI. To see how it works, have a look at the source code.

What is still missing is the ability to *like* messages. As the client already provides buttons for this, we are going to handle their `click` events. For performance reasons this is done once for the list, not for each button individually. Of course, then you need to get the `id` of the message whose button was clicked.

Finally, you can run the `like` command for the message of your choice:

```
chat.lists.messages.readAndObserve().
  // ...

document.querySelector('.messages').
  addEventListener('click', event => {
    if (!event.target.classList.
      contains('likes')
    ) {
      return;
    }

    const messageId = event.target.
      getAttribute('data-message-id');

    chat.communication.message(messageId).
      like().
      failed(err => console.error(err));
  });
```

Let's chat!

Now, start your wolkenkit backend by running the following command from inside the `chat` directory, and wait until a success message is shown:

```
$ wolkenkit chat
```

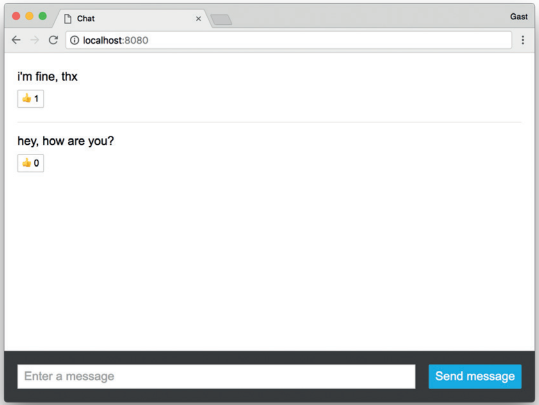
wolkenkit only takes care of the server part of your application and does not run the client for you automatically. Hence you need to install an HTTP server and run the client manually. We are using *http-server* that can easily be installed by using the following command:

```
$ npm install -g http-server
```

Once you have done that run the client using the following command. This will automatically launch a browser and open the client:

```
$ http-server ./client/ -o
```

You are now able to chat. This even works with multiple browsers concurrently:



i'm fine, thx

👍 1

hey, how are you?

👍 0

Let's recap what you have achieved:

- Users can send messages
- Users can like messages
- Sent messages are visible to all users
- When a user enters the chat they are shown the previously sent messages

- When a user receives a message the UI is updated in real-time
- Sending and receiving messages is possible using an API
- Sending and receiving messages is encrypted

Yay, congratulations!

You have created your first application from scratch, including a real-time client! I hope that you will have a great time with wolkenkit. If you would like to chat, feel free to contact my colleagues at **www.thenativeweb.io**

Love, Sophie

Your notes on wolkenkit

We are **wolkenkit**

Thank you very much for your interest in wolkenkit! We hope that you like it and find it useful.

In case of questions, ideas, or any other feedback, feel free to contact us. Say **hello@thenativeweb.io**

the native web.

Why **wolkenkit**

Software development is not an end in itself. Instead, software gets written to solve actual real-world problems. Unfortunately, this fails regularly for various reasons.

That‘s why we have built wolkenkit, a semantic JavaScript backend that addresses three important aspects of software development, and that empowers you to build better software faster.

www.wolkenkit.io

the native web.