

## Connecting a client



Install the wolkenkit SDK into your application:

```
$ npm install wolkenkit
```

To connect to a wolkenkit application, use the following example:

```
const wolkenkit = require('wolkenkit');

wolkenkit.connect({ host: '...' }).
  then(app => { /* ... */ }).
  catch(err => { /* ... */ });
```

Select a port other than 443 using the `port` property.

## Sending commands



```
app.communication.message(id).send({
  text: 'hey, how are you?'
}).
  failed((err, command) => {
    // ...
  }).
  delivered(command => {
    // ...
  }).
  await('sent', (event, command) => {
    // ...
  }).
  timeout('30s', command => {
    // ...
  });
```

To create a new aggregate, drop the `id`. All callbacks are optional. If provided, `failed` should come first. Use an array to `await` multiple events.

## Receiving events



```
app.events.observe({
  where: { name: 'sent' }
}).
  failed(err => { /* ... */ }).
  started(cancel => { /* ... */ }).
  received((event, cancel) => { /* ... */
});
```

The `where` clause, and the `failed` and `started` callbacks are optional. If provided, `failed` should come first.

## Reading lists



```
app.lists.messages.read({
  where: { /* ... */ },
  orderBy: { /* ... */ },
  skip: /* ... */,
  take: /* ... */
}).
  failed(err => { /* ... */ }).
  finished(messages => { /* ... */ });
```

All criterias and `failed` are optional. If provided, `failed` should come first. Use `readOne` instead of `read` to read a single record. Use `readAndObserve` with the same options to retrieve real-time updates:

```
app.lists.messages.readAndObserve().
  failed(err => { /* ... */ }).
  started((messages, cancel) => {
    // ...
  }).
  updated((messages, cancel) => {
    // ...
  });
```

## Quick start guide and cheat sheet

### Your semantic JavaScript backend

Setup an API for your business to bridge the language gap between your domain and technology. Map your domain of knowledge to semantic JavaScript code in no time.

The result: One language, one code. Build better software faster to solve real-world problems.

Want to get started?  
<https://www.wolkenkit.io>



wolkenkit

## Using the CLI

To start an application, run the following command:

```
$ wolkenkit start
```

You may specify a port using the `--port` flag. To stop an application, run:

```
$ wolkenkit stop
```

To restart an application, e.g. after you have updated its code, run:

```
$ wolkenkit reload
```

To restart an application as well as its infrastructure services, run:

```
$ wolkenkit restart
```

To verify whether an application is running, use:

```
$ wolkenkit status
```

## Storing data permanently

Any data will be destroyed when stopping an application. Set a shared key to store data permanently:

```
$ wolkenkit start --shared-key <secret>
```

Provide the same shared key whenever you restart the application. To destroy any stored data, run:

```
$ wolkenkit stop --dangerously-destroy-data
```

Use the `WOLKENKIT_SHARED_KEY` environment variable to store the shared key permanently.

## Using environments

To use a specific environment, use the `--env` flag:

```
$ wolkenkit start --env <environment>
```

Alternatively, set `WOLKENKIT_ENV` appropriately.

## Installing wolkenkit

On macOS and Linux, install Docker and Node.js, then run:

```
$ curl https://install.wolkenkit.io | bash
```

You may need to enable VT-x for Docker to work.

## Updating wolkenkit

```
$ wolkenkit update
```

## Initializing an application

Using the default template:

```
$ wolkenkit init
```

Using a custom template:

```
$ wolkenkit init --template  
git@github.com:<org>/<repo>.git
```

Suffix with #<branch-or-tag> to not use master.



### In case of questions...

...ask on StackOverflow using the #wolkenkit tag.

You can also contact my colleagues at the native web via mail, and say [hello@thenativeweb.io](mailto:hello@thenativeweb.io).

## Defining aggregates

```
const initialState = {  
  text: '',  
  isAuthorized: {  
    commands: {},  
    events: {}  
  }  
};  
  
const commands = {};  
const events = {};  
  
module.exports = {  
  initialState, commands, events };
```

When extending the initial state, you may use any JSON-compatible data type.

## Defining commands

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

Each code path must either call `mark.asDone()` or `mark.asRejected()`. One command may publish one or more events. Event data is optional. Use an array to run multiple command handlers in series.

## Defining events

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

## Configuring authorization

```
isAuthorized: {  
  commands: {  
    send: {  
      forAuthenticated: true,  
      forPublic: false  
    }  
  },  
  events: {  
    sent: {  
      forAuthenticated: true,  
      forPublic: true  
    }  
  }  
}
```

By default, only the owner is authorized. Use the aggregate's `transferOwnership` and `authorize` functions to change authorization at runtime.

### Write model

The write model handles commands, and stores and publishes events using the domain.

### Read model

The read model handles events and transforms them into lists.

## Defining lists

```
const fields = {  
  text: { initialState: '' }  
};  
const when = {};  
  
module.exports = { fields, when };
```

Set `fastLookup` to `true` to index a field, set `isUnique` to `true` to mark a field as unique.

## Handling events

```
'communication.message.sent' (  
  messages, event, mark  
) {  
  messages.add({  
    text: event.data.text  
  });  
  mark.asDone();  
}
```

Lists inherit the events' authorization. Use the list's `transferOwnership` and `authorize` functions to change authorization at runtime.

To update and to remove list items, use the following template. The syntax is similar to MongoDB:

```
messages.update({  
  where: { /* ... */ },  
  set: { /* ... */ }  
});  
  
messages.remove({  
  where: { /* ... */ }  
});
```