CMDI Maker – Entwicklerhandbuch

v1.1

Inhalt

[Paradigmen 1](#_Toc419712651)

[Formulare 1](#_Toc419712652)

[Eigenschaften eines Feldes 2](#_Toc419712653)

[Environments 3](#_Toc419712654)

[Laden eines Environments 4](#_Toc419712655)

[Good Practices beim Erstellen eins Environments 4](#_Toc419712656)

[Save and Recall 5](#_Toc419712657)

[Abhängigkeiten 5](#_Toc419712658)

# Paradigmen

Der CMDI Maker soll für den Benutzer so einfach wie nur irgendwie möglich zu bedienen sein. Er soll ihm dabei sinnvolle und nützliche Funktionen anbieten, die ihn schneller ans Ziel führen: Fertige Metadaten.

Das Benutzerinterface ist immer freundlich und einladend zu gestalten. Das Design ist flach, auf ablenkende 3D-Elemente werden verzichtet. Dafür werden kräftige Farben eingesetzt, um einschlägiger Software im Bereich der Korpus-Linguistik eine Alternative zu bieten und um technisch wenig versierten Nutzern den Eindruck der Einfachheit zu vermitteln.

# Formulare

Die Grunddatenstruktur, die beim CMDI Maker zum Einsatz kommt, sind Formulare (Forms). Die App kann Formulare aus Form Templates generieren. Diese Templates beschreiben im JSON-Format, wie ein Formular auf dem Bildschirm aufgebaut sein wird. Aus den Templates ergibt sich auch die Datenstruktur, in der Formulardaten gespeichert werden.

Beispiel für ein Form Template:

title: "Contact",

name: "contact",

type: "column",

fields: [

{

heading: "Name",

name: "name",

type: "text"

},

{

heading: "Address",

name: "address",

type: "text"

},

{

heading: "Email",

name: "email",

type: "text"

}

]

Bei dieser Formularstruktur werden die Daten die der CMDI Maker speichert, entsprechend so formatiert sein:

"contact":   
{  
 "name": "Sebastian Zimmer",

"address: "Universitätsstr. 22",  
"email": "me@example.com"

}

Die APP bietet 4 Funktionen an, die für Formulare genutzt werden können:

1. **make (parent, field, element\_id\_prefix, element\_class\_prefix, data\_object)**

Diese Funktion erstellt ein Formular im DOM, nach dem Template „field“. Das Formular wird auf der Website unter dem DOM-Node „parent“ eingehängt. Die IDs aller Elemente im DOM bekommen das Präfix „element\_id\_prefix“, die Klassen das Präfix „element\_class\_prefix“. Optional kann mit „data\_object“ ein Datenobjekt übergeben werden, dessen Daten im Formular erscheinen sollen.

Falls das Template Felder des Types „special“ enthält, wird die Funktion „on\_special“ aufgerufen. Ihr wird das entsprechende Feld sowie „element\_id\_prefix“, „element\_class\_prefix“ und „parent“.

1. **fill (field, element\_id\_prefix, data\_object, on\_special)**

Diese Funktion füllt ein bestehendes Formular, welches nach dem Template „field“ erstellt worden ist, mit Daten eines Datenobjekts „data\_object“. Damit die Funktion weiß, wo sich das Formular im DOM befindet, benötigt sie die Angabe des ID-Präfixes der DOM-Elemente, „element\_id\_prefix“.

Falls das Template Felder des Types „special“ enthält, wird die Funktion „on\_special“ aufgerufen. Ihr wird das entsprechende Feld sowie „element\_id\_prefix“ und „parent“.

1. **createEmptyObjectFromTemplate (field)**

Diese Funktion erstellt ein Datenobjekt nach Vorlage eines Form-Templates „field“ und gibt dieses zurück.

1. **fillObjectWithFormData (object, element\_id\_prefix, field, on\_special)**

Diese Funktion überprüft die Eingaben, die in einem Formular nach dem Template „form\_element“ mit dem ID-Präfix „element\_id\_prefix“ und füllt ein Objekt „object“ mit diesen Daten.

Bei den vorliegenden Funktionen ist mit der Variable „field“ das Template gemeint, das beschreibt, wie ein Formular aufgebaut ist. Durch die rekursive Natur der Funktionen ist es nicht sinnvoll, der Variable „field“ den auf den ersten Blick vllt. Intuitiveren Namen „template“ zu geben. Die Funktionen rufen sich nämlich selbst immer wieder auf und dabei verändert sich das, was in „field“ übergeben wird. Ein Field kann u. a. eine komplette Formularvorlage, die Spalte einer Form, ein Sub-Bereich innerhalb der Form oder ein einzelnes Element wie ein Eingabefeld sein.

## Eigenschaften eines Feldes

* title: Relevant bei Columns und Subareas. Hier kann man eine Überschrift für diese angeben.
* Type: Spezifiziert den Typ des Feldes. Mögliche Werte sind [form, column, subarea, text, textarea, date, year, checkbox, special]
* name: Name des Feldes, das auch als Key in der Datenstruktur verwendet wird
* heading: Überschrift des Eingabefeldes
* allowed\_chars: String mit allen Zeichen, die im Eingabefeld erlaubt sind
* not\_allowed\_chars: String mit Zeichen, die im Eingabefeld nicht erlaubt sind
* fields: Relevant für Felder des Typs form, column oder subarea. Fields ist ein Array, in welchem weitere (Sub-)Felder spezifiziert sind.

# Environments

Environments sind Arbeitsumgebungen, die sich dem Benutzer als Profiles darstellen. Er kann in den Einstellungen und auf der Startseite zwischen verschiedenen Profiles eines auswählen. Jedes Environment verfügt über einen eigenen Workflow mit eigenen Modulen, welche beim Laden des Environments in der Titelleiste dargestellt werden:



Jedes der Workflow-Module kann über eigene Funktionen verfügen, die in der Funktionsleiste angezeigt werden:



Jedes Modul verfügt über ein sog. Identity-Objekt, welches von der App verwendet wird, um bspw. den Workflow in der Titelleiste darzustellen.

Bsp.:

my.identity = {

id: "actor",

title: "Actors",

icon: "user"

};

Das Identity-Objekt muss eine eindeutige ID, einen Titel und ein Icon haben.

Ein Environment kann weiterhin Module besitzen, welche nicht zum Workflow dazugehören. Es ist dann die Aufgabe des Environments, diese Module zu initialisieren und zu nutzen.

Jedes Environment kann außerdem Settings exportieren, die in der View „Einstellungen“ bei geladenem Environment zu sehen sind. Die Settings sind im Hauptmodul des Environments definiert:

my.settings = function(){

return [

{

title: my.l("settings","output\_format"),

id: "output\_format\_select",

type: "empty"

},

{

title: my.l("settings","cmdi\_metadata\_creator"),

description: my.l(

"settings",

"cmdi\_metadata\_creator\_description"

),

type: "text",

name: "metadata\_creator",

id: "metadata\_creator",

value: "CMDI Maker User"

},

…

Settings können einen der folgenden Typen haben: radio, select, toggle, file, text, empty, link

Das Settings-Objekt eines Environments kann auch (wie im Beispiel) von einer Funktion namens settings zurückgegeben werden, z. B., wenn die Termini sprachabhängig sind.

## Laden eines Environments

Der Ablauf beim Laden eines Environments wurde bewusst so festgelegt:

1. Die Settings des Environments werden initialisiert (zu finden im Hauptmodul des Environments mit dem Key „settings“)
2. Beim Erstellen des Workflows werden pro Modul zuerst seine Funktionen geladen, die das Modul im Key „functions“ definiert.
3. Dann wird das Modul initialisiert, d.h. falls es eine Funktion unter dem Key „init“ aufweist, wird diese ausgeführt. Dabei wird der Parameter „view“ übergeben, welcher das DOM-Element für die Ansicht („View“) des Moduls referenziert. Das Modul kann bei Bedarf mit seiner Init-Methode darauf zugreifen und Veränderungen an der View durchführen.

Da zuerst die Funktionen für ein Modul erstellt werden, kann sich das Modul bei der Call von module-init(view) darauf verlassen, dass diese zur Verfügung stehen und bei Bedarf nochmal angepasst werden.

## Good Practices beim Erstellen eins Environments

* Hauptskript, Generator-Logik, Formular-Modelle, Workflow-Items, LanguagePacks jeweils in eigene JS-Dateien
* Alle Dateien (sowohl JS als auch CSS) mit einem Präfix, bestehend aus Environment-Name und Unterstrich, versehen
* Verwendung des Javascript Module Patterns für Hauptskript, Generator und Workflow-Items. (Siehe: <http://www.adequatelygood.com/JavaScript-Module-Pattern-In-Depth.html>)
* Alle DOM-Klassen und DOM-Element-IDs und damit auch alle CSS-Selektoren sollten ebenfalls mit einem Präfix, bestehend aus Environment-Name und Unterstrich, beginnen
* Bsp. für Dateien in einem Environment:
  + lido\_main.js (Hauptskript)
  + lido\_forms.js (enthält Formular-Modelle)
  + lido\_1.js (erstes Workflow-Item)
  + lido\_2.js (zweites Workflow-Item)
  + lido\_3.js (drittes Workflow-Item)
  + lido\_4.js (viertes Workflow-Item)
  + lido\_5.js (fünftes Workflow-Item)
  + lido\_output.js (sechstes und letztes Workflow-Item, das die Generierung des XMLs anstößt und die Resultate anzeigt)
  + lido\_generator.js (Generator-Skript, das das XML mit Hilfe der Bibliothek XMLStrings generiert)
  + lido\_LanguagePacks.js (enhält LanguagePacks)
  + lido\_styles.css (Stylesheet)
* Jedes Workflow-Item muss über eine einmalige ID verfügen und sollte eines der verfügbaren Icons verwenden. Die Icons befinden sich im CMDI-Maker-Repository unter src/img/icons

# Save and Recall

Damit die Eingaben, die der Nutzer im CMDI Maker tätigt, nicht verloren gehen, nachdem das Browser-Fenster geschlossen wird, können diese in einer Browser-Datenbank (LocalStorage) gespeichert werden. Bei einem Speichervorgang speichert die App sowohl ihren eigenen aktuellen State (dazu gehören z. B. die zurzeit aktive View, das aktuell geladene Environment oder Einstellungen wie das Intervall für die automatische Speicherung) als auch den State des derzeit aktiven Environments. Dazu fragt die App bei jedem Speichervorgang den State des Profils und all seiner Workflow-Module ab.

Um diese Funktion der Speicherung von Daten zu nutzen, benötigt das Hauptskript, sowie jedes Workflow-Modul zwei Methoden:

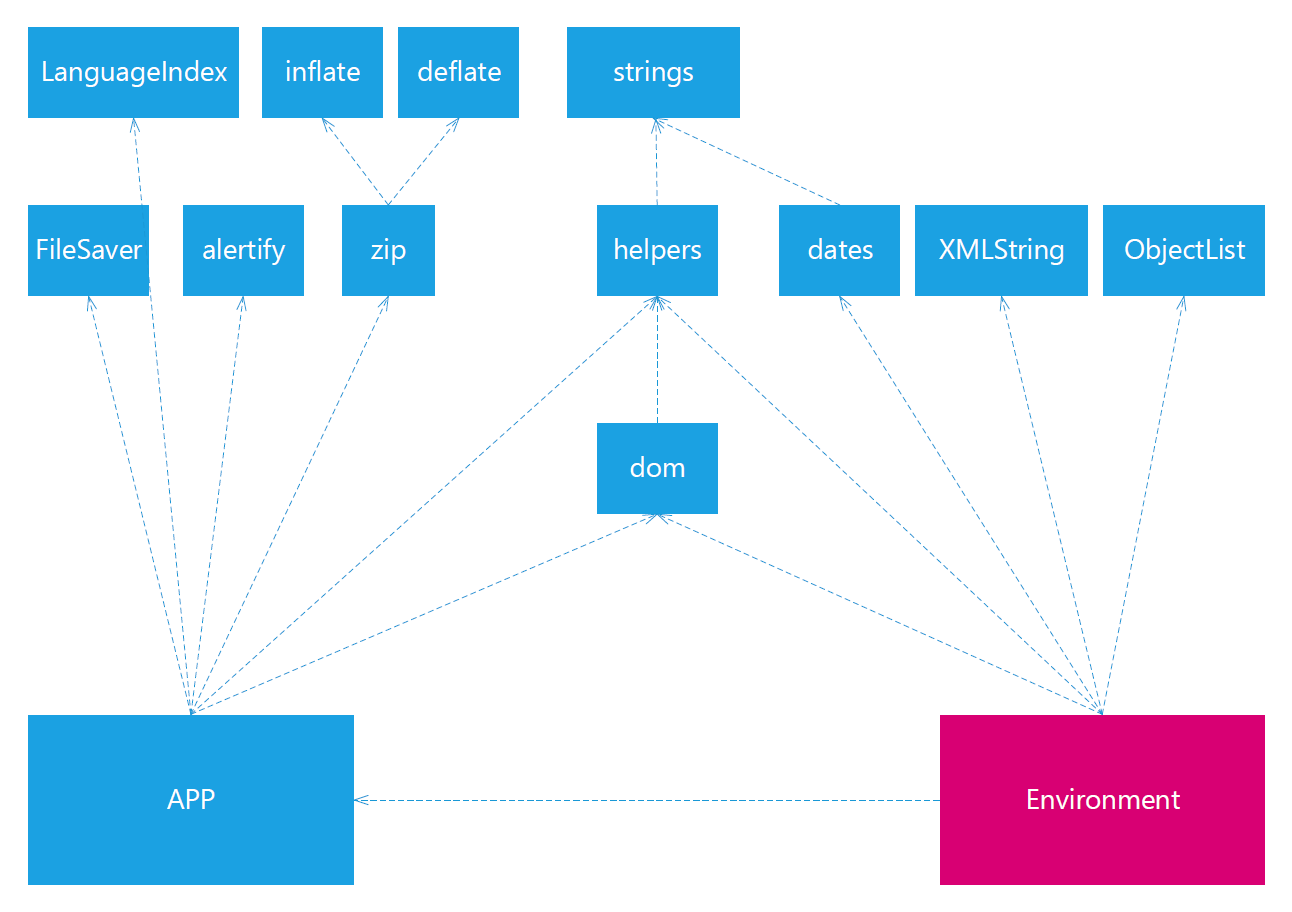
1. **my.getSaveData()**

Diese Funktion wird von der App bei einem Speichervorgang aufgerufen und sollte den aktuellen State des Moduls als ein Objekt zurückgeben.

1. **my.recall(data)**

Diese Funktion wird von der App bei einem Ladevorgang aufgerufen und als Parameter wird das Objekt übergeben, das von diesem Modul vorher zur Speicherung freigegeben wurde. Das Modul ist dann dafür verantwortlich, seinen State auf der Grundlage dieser Daten wiederherzustellen.

# Abhängigkeiten



CMDI Maker Dependency Diagram

Man kann den Code einer CMDI Maker-Instanz in 3 große Teile einteilen:

* APP CORE
* Externe Libraries
* Environments

Zu den externen Libraries gehören

* zip.js (<http://gildas-lormeau.github.io/zip.js/>, wiederum abhängig von deflate.js und inflate.js)
* alertify.js (<http://fabien-d.github.io/alertify.js/>, Achtung: Diese Version von Alertify wird nicht mehr weiterentwickelt, es gibt jedoch einige vielversprechende Forks. Diese haben jedoch leichte Änderungen in ihrer API und wurden bisher nicht im CMDI Maker implementiert)
* FileSaver.js (<https://github.com/eligrey/FileSaver.js>)
* helpers (Im Rahmen des Projekts CMDI Maker entwickelt)
* dom (Im Rahmen des Projekts CMDI Maker entwickelt)
* dates (Im Rahmen des Projekts CMDI Maker entwickelt)
* strings (Im Rahmen des Projekts CMDI Maker entwickelt)
* XMLString (Im Rahmen des Projekts CMDI Maker entwickelt)
* ObjectList (Im Rahmen des Projekts CMDI Maker entwickelt)

Weiterhin ist die Datei LanguageIndex.js zu den externen Quellen hinzu zu zählen, die ein Javascript-Array von ISO639-3-Sprachcodes enthält. Diese wird von der APP benötigt, um den Environments eine Sprachsuch-Funktion anbieten zu können.

Nicht nur der APP Core, sondern auch die Environments haben Zugriff auf die externen Libraries, da sich diese im globalen Namespace von Javascript befinden. Dadurch kann das Environment XML-Strings erstellen, seine Objekte mit ObjectList verwalten, etc. Für manche Dinge bietet der APP Core aber einen bequemen Wrapper an, der anstelle der Library-APIs verwendet werden sollte. Dazu gehören:

|  |  |  |
| --- | --- | --- |
| **Beschreibung** | **Betroffene Librarys** | **Wrapper-Funktionen** |
| Generieren von Alerts, Confirms und Log Messages am unteren rechten Rand | alertify.js | APP.log, APP.confirm, APP.alert |
| Speichern („downloaden“) einer Datei, die Text enthält. | FileSaver.js | APP.saveTextfile |
| Aus allen Texten, die das Environment in einer XML-Textarea darstellt, jeweils eine Datei generieren, diese Dateien dann in einem ZIP-Archiv komprimieren und die ZIP-Datei anschließend runterladen | zip.js / FileSaver.js | APP.zipAllOutputFiles |
| Aus allen Texten, die das Environment in einer XML-Textarea darstellt, jeweils eine Datei generieren, und jede dieser Dateien anschließend einzeln auf dem Computer speichern | FileSaver.js | APP.saveAllOutputFiles |
| Sprachsuche | LanguageIndex.js | APP.doStandardLangaugeSearch |

Es kann durchaus sein, dass der APP Core nicht alle Funktionalitäten einer Library über einen Wrapper anbietet. In diesen Fällen muss das Environment direkt auf die Library zugreifen.