# Dokumentation Alarm

# Projektbeschreibung

ALARM observed and scenario climate data

Assessing Large scale Risks for biodiversity with tested Methods

www.alarmproject.net/climate/climate/

The ALARM climate data are made available free of charge for non-commercial uses provided that the sources are properly acknowledged: The ALARM climate observations and scenarios are described by Fronzek et al. (2011), with additional methodological information provided in Mitchell et al (2004). The data are downloadable from: www.alarmproject.net/alarm

# Projektablauf

1. *Einrichtung Entwicklungsumgebung*

Das Projekt wurde mit MongoDB als Datenbank-Modell und Rails 3.1 als Backend umgesetzt. Dazu mussten einige Schritte getätigt werden.

- Installation von MongoDB auf Mac OS X und Ubuntu 10.10
- Einrichtung eines neuen Git-Repositories für die Entwicklung (https://github.com/lpsBetty/alarmproject)

Im Projekt selber wurde mit TextMate (Mac OS X) und Aptana Studio 3 (Ubuntu) gearbeitet.

2. *Recherche*

Vorab wurde entschieden das Projekt mithilfe von MongoDB und Rails 3.1 umzusetzen. Da wir bisher, außer theoretischem Wissen aus Vorträgen bzw. Vorlesungen, keinerlei praktische Erfahrung mit MongoDB hatten, mussten wir uns informieren und hatten eine längere Recherchephase. Es wurden Grundlagen von Mongo selber nachgelesen und recherchiert. Vor allem wurden Informationen gesammelt, die uns MongoDB in Kombination mit Rails besser erläuterten.

Uns stellte sich die Frage was für einen Object Mapper wir für Ruby für MongoDB einsetzen sollten. Es standen folgende zur Verfügung:

- MongoMapper
  - http://mongomapper.com/
  - https://github.com/jnunemaker/mongomapper
- Mongoid
  - http://mongoid.org/
  - https://github.com/mongoid/mongoid

Da beide dieser Tools laufend aktualisiert und verbessert werden, ist es uns nicht alleine aufgrund von der „Aktualität“ der einzelnen Object Mapper leicht gefallen eine Entscheidung zu fällen. MongoMapper existiert bereits länger, wird aber genauso wie Mongoid maintained.

Nach einigen Recherche-Einheiten haben wir uns für den Einsatz von Mongoid als Object Mapper in unserem Projekt entschieden. Entscheidungshilfen waren unter anderem folgende Blogeinträge bzw. Informationsquellen:

- http://www.rubyinside.com/mongoid-vs-mongomapper-two-great-mongodb-libraries-for-ruby-3432.html

- http://stackoverflow.com/questions/1958365/mongoid-or-mongomapper

## 3. Start

Nach der Recherche und Überzeugung, dass wir Mongoid als Object Mapper benutzen wollten begann die Umsetzungsphase:

- Neues Rails 3.1.3 Projekt
- Mongoid
- Mongo-Server starten:
  - Händisch: mongod –dbpath=db (im Rails-Projektordner)
  - Rake-Task: https://gist.github.com/167620
  - Mongo-Console (systemabhängig) mit: sudo start mongodb

## 4. Entwicklung Datenbank-Modell

Die Entwicklung des Datenbank-Modells war für uns anfangs nicht leicht, da wir keinerlei Erfahrung mit einem guten Aufbau einer MongoDB hatten bzw. immer noch nicht haben. Unsere ersten Entwürfe wären wahrscheinlich für eine relationale Datenbank, aber nicht für eine dokumentenbasierte, gut gewesen. Am Ende haben wir uns für folgendes Schema entschieden:

- Model
  - field :name, type: String
  - field :desc, type: String
  - has_many :values
- Point
  - field :x, type: Float
  - field :y, type: Float
  - has_many :values
- Scenario
  - field :name, type: String
  - field :desc, type: String
  - has_many :values
- Variable
  - field :name, type: String
  - field :desc, type: String
  - has_many :values
- Value
  - field :year, type: Integer
  - field :month, type: Integer
  - field :number, type: Float
  - belongs_to: :model
  - belongs_to : :point
  - belongs_to :scenario
  - belongs_to :variable

Dieses Schema scheint auf den ersten Blick sehr relational. Wir haben uns aber trotzdem entschieden das Projekt mit diesem Schema umzusetzen. Aus Erfahrungen kann man sehr viel für die Zukunft lernen.

## *5. Entwicklung Datenbank-Importer*

Der nächste wichtige Schritt war die Entwicklung eines Datenbank-Importers um die vorhandenen JSON-Files in unser Schema zu portieren.

Wir haben hierfür eine eigene Klasse „Importer“ geschrieben, welche Methoden für die Schema-Portierung bereitstellt. Zu finden ist die Klasse unter „lib/tasks/importer/importer.rb“.

Mit der Zeit sind die unterschiedlichsten Varianten des Importers entstanden. Die Basics wurden relativ schnell entwickelt. In der Folge mussten wir allerdings die Performance stets verbessern.

Einige Daten und Fakten:

*Anfängliche Versionen:*
In der Version1 hatten wir noch ziemlich viele Rails-spezifische Funktionen im Importer integriert. Beispielsweise initialisierten wir neue Objekte mit Record.**new.** Eine Hochrechnung nach ersten dem Import von ein paar Test-Daten, welche einen Bruchteil der gesamten Daten ausmachten, hätten wir über 60 Stunden für den kompletten Import benötigt. Die initialisierten Objekte wurden sofort in Dokumente umgewandelt bzw. direkt erstellt. Dieser direkte Erstellungsprozess hat sehr lange gedauert.

*Nach Änderungen:*
Durch die Hochrechnung aus der ersten Version kamen wir zum Entschluss, dass wir die Dokumente nicht direkt während des Imports erstellen können, da dies zu viele Ressourcen benötigt und insgesamt einfach zu lange braucht. Wir entschieden uns die Objekte in JSON-Objekte umzuwandeln und in ein JSON-File zu schreiben. Dieses JSON-File sollte anschließend direkt mit ‚mongoimport –d „datenbank-name“ –c „collection-name“ „filename“‘ importiert werden. Der Zeitgewinn war bereits enorm. So ergab eine nächste Hochrechnung bereits einen Zeitgewinn von fast 1/3 ü viel, aber insgesamt doch noch deutlich zu lange.

*Weitere Änderungen:*
In weiteren Änderungen splitteten wir das eine generierte JSON-File in mehrere „kleinere“ Files auf. Dies brachte wieder einen Performance-Gewinn. Eine besonders hohe Performance-Steigerung brachte die direkte Erstellung von JSON-Objekten und damit die Umgehung von Record.**new.**

*Refactoring und weitere Schritte:*
In der Folge wurden noch weitere Refactoring-Schritte und Performance-Steigerungen getätigt. Wir denken, dass wir für unser Datenbank-Schema schlussendlich die bestmögliche Importlösung durch weiteres Refactoring programmiert haben. Wir haben beispielsweise noch eine parallele Lösung für den Import von mehreren Scenarios gleichzeitig eingebaut. Dabei nutzen wir ein Rails-gem namens „parallel“. Dieses Gem kann für jede vorhandene CPU bzw. für jeden Core ein Scenario importieren. Sprich sollte ein Rechner über 8 CPUS verfügen, so können auch 8 Files gleichzeitig importiert werden.

*Spezielles:*
Um Umrechnungsfehler bei der Multiplikation mit den Faktoren für die Value-Werte haben wir BigDecimal eingesetzt. Mithilfe von BigDecimal lassen sich Integer-Float Rundungsfehler besser behandeln. (z.B. nicht 12,00001, sondern 12,1)

*Resümee Import*
Unser größtes Problem war die große Masse an Daten. Durch unser (zugegeben nicht wirklich gutes) Datenbank-Schema haben wir sehr viele Daten generieren müssen. Alleine die JSON-Files, welche im ersten Schritt eines Imports erstellt werden, brachten am Ende insgesamt über 76 GB (3 Scenarios, 3 Variablen) auf die Waage.
Die Datenbank hat zu Ende eine Größe von knapp 70 GB erreicht. Dabei sind Indexes noch nicht eingerechnet. Diese benötigen zusätzlich noch ca. 20 GB.
Der Import selber hat für jedes Scenario am Ende ca. 3-5h gedauert. Wir können dies leider nicht punktgenau sagen, da wir den Import oftmals mit Unterbrechungen durchgeführt haben.

Zeitbeispiele:

'BAMBU.A2.HadCM3.2001-2100.pre' converted into json for import in 5189.81 s
Convert finished in 5190.37 s

'BAMBU.A2.HadCM3.2001-2100.gdd' converted into json for import in 5162.37 s
Convert finished in 5162.96 s

'BAMBU.A2.HadCM3.2001-2100.tmp' converted into json for import in 5425.33 s
Convert finished in 5427.76 s

'GRAS.A1FI.HadCM3.2001-2100.gdd' converted into json for import in 5913.0 s
Convert finished in 5914.77 s

'GRAS.A1FI.HadCM3.2001-2100.pre' converted into json for import in 16074.93 s
Convert finished in 16077.29 s

Anhand dieser Zeitbeispiele ist auch zu erkennen, dass nicht jedes File gleich lang gebraucht hat. Beispielsweise hat beim letzten File der Arbeitsspeicher zu swappen begonnen.

Wir haben den Importer getestet und er hat eigentlich soweit auch funktioniert. Trotzdem sind wir im Nachhinein nach einem vollständigen Import noch auf einen Fehler gestoßen (Datentyp string statt integer). Solche Fehler sind sehr störend während des Arbeitens und erschweren die Projektarbeit.

Probleme beim Import nochmals zusammengefasst:
- Sehr viele generierte Daten
- Große Datenbank
- Arbeitsspeicherprobleme beim Import (Laptops sollten nicht für solche Arbeiten benutzt werden)
- Festplattenspeicher: teilweise Platzprobleme durch viele Daten
- Paralleler Import nur teilweise möglich ü Arbeitsspeicher wird auf Laptops knapp

*Ergebnis:*
Rake-Tasks:

- rake db:get_data: Generiert JSON-Files für das Schema
- rake db:import: Importiert generierte JSON-Files in Datenbank
- rake db:full_import: fasst die oberen zwei Rake-Tasks zusammen

Datenbank von knapp 90 GB (350.000.000 Dokumente)

Der komplette Import-Prozess hat am Ende einfach zu lange gedauert. Durch auftretende Probleme und später gefundene Fehler wurde dies noch weiter verzögert.

## 6. *Eigentliche Umsetzung*

### a. Technischer Aufbau

*Models:*
Das Projekt stützt sich auf 7 Models. Darunter zählt das Model „Model“, welches eigentlich nur das Model beinhaltet. In unserem Fall ist das „Europe“. „Scenario“ beinhaltet das Scenario (z.B. „BAMBU“), „Variable“ (z.B. „tmp“, „pre“, „gdd“). Das „Point“ Model beinhaltet x und y Werte einer Koordinate auf der Map. Die bisher aufgezählten Models stehen alle in 1:n Beziehung mit dem Model „Value“.

„Value“ besitzt zusätzlich noch die Attribute „year“, „month“ und „number“. „number“ ist der Wert einer x und y Koordinate eines Punktes. Zwecks der Übersicht haben wir vom Model „Value“ noch „Map“ und „Prop“ abgeleitet und die Struktur aufgesplittet.

„Value“ stellt private Methoden für „Map“ und „Prop“ zur Verfügung. Die eigentlichen Berechnungen und Datenabfragen werden mit der Hilfe von Methoden von „Map“ und „Prop“ gelöst.

*Controllers:*
Wir benutzen für unser Projekt einen Controller. Der „ValuesController“ wird über die Routes entsprechend angesteuert und leitet die gewünschten Abfragen an die betreffenden Model-Methoden weiter.
Beispiels-Route:

```
  match 'mapval/:model/:scenario/:year/:month/all' => 'values#mapval', :month => /\d+/
```

Mithilfe dieser Route wird die mapval all Abfrage an die dementsprechende Controller-Action weitergeleitet und in Folge dann das Ergebnis geliefert.

Wir haben stark auf die neuen Features vom Rails3-Routing gesetzt um die Controller so sauber wie möglich halten zu können.

*Views:*
Wir verwenden für die Ausgabe außer einem Layout eigentlich keinerlei Views, da der Schwerpunkt in diesem Projekt auf die technische und nicht gestalterische Umsetzung gelegt wurde. Ausnahme davon ist die REST-Schnittstelle. Diese wird allerdings später noch kurz erläutert.

*Map Reduce:*
Durch weitere Recherche fanden wir heraus, dass MongoDB eine eingebaute map/reduce Funktion hat, die besonders für die schnelle Berechnung von Aggregationen geeignet ist: http://www.mongodb.org/display/DOCS/MapReduce

Da die Werte meist nach dem Punkt (Model: Point) gruppiert werden musste, war es mit dieser Funktion einfacher zu bewerkstelligen als mit dem Rails-Äquivalent *group_by*.

### b. Testing

Wir haben das Projekt mithilfe vom Rspec-Rails gem (https://github.com/rspec/rspec-rails) getestet. Zum einen wurde der Importer selber getestet und zum anderen haben wir functional Tests geschrieben um die zurückgegebenen Ergebnisse überprüfen zu können. Mit diesen functional Tests werden gleichzeitig die Methoden in den Models mit getestet. Auf explizite Unit-Tests haben wir bewusst verzichtet, da bei diesem Projekt der Output stimmen muss und dieser mit den functional Tests gut abgedeckt ist.

### c. REST-Schnittstelle

Wir haben in unser Projekt eine REST-Schnittstelle eingebaut um Daten direkt im Browser manipulieren zu können. Diese Schnittstelle stützt sich auf das Typus Gem (https://github.com/fesplugas/typus). Das Typus Gem stellt von Haus aus bereits CRUD Methoden bzw. Actions zur Verfügung um Ressourcen ändern zu können.

Hierbei war für uns das einzige Problem, dass Typus relationale ORMs unterstützt und dokumentenbasierte noch nicht zu 100%. Typus unterstützt zwar seit kurzem Mongoid, alle Funktionen sind allerdings noch nicht umgesetzt bzw. vollständig integriert. Dadurch konnten wir Typus nicht einfach so übernehmen, sondern mussten es ein wenig tweaken um unser Projekt damit zum Laufen zu bringen. Nun ist es Möglich alle Ressourcen neu zu erstellen, zu ändern oder zu löschen. Einzig die „Values“ können nicht geändert oder gelöscht werden.

Sollte es von Nöten sein, dass Values geändert oder gelöscht werden sollen, muss auf die Standard-Rails-Konsole zurückgegriffen werden. Diese kann wie bekannt mit „rails c“ gestartet werden.

Zusätzlich zum Typus-Backend und der Rails-Konsole steht die MongoDB-Konsole zur Verfügung. Diese kann mit „mongo“ gestartet werden. Weitere Informationen zur MongoDB-Konsole sind unter http://www.mongodb.org abrufbar.

# Erfahrungen

## 1. Allgemeines

Mit diesem Projekt haben wir die erste Erfahrung mit MongoDB/Mongoid mit Rails gemacht. Zusätzlich haben wir das erste Mal mit einer solchen Menge an Daten gearbeitet.

Uns ist leider zu spät aufgefallen, dass unser Denkansatz betreffend Datenbank-Schema leider zu relational ausgefallen ist. Unser Datenbank-Schema wäre für eine relationale Datenbank wie MySQL eindeutig besser zu gebrauchen gewesen, wie für eine MongoDB Datenbank. Dadurch sind die Abfragezeiten auch sehr lange. Die Abfragen funktionieren mit weniger Daten gut und auch richtig. Rein technisch haben wir somit gute Arbeit geleistet um zu den richtigen Ergebnissen zu kommen. Unser Projekt lässt sich leider nicht skalieren, was extrem schade ist. Für uns war die Arbeit allerdings nicht umsonst, wir haben sehr viel gelernt. Einerseits, dass genau bei so heiklen Themen wie einem Import von Daten sehr genau Acht gegeben werden muss und auch jedes noch so kleinste Detail getestet sein sollte (Stichwort falscher Datentyp). Des Weiteren haben wir den generellen Umgang mit MongoDB kennen gelernt und konnten uns in diesem Bereich weiter entwickeln.

## 2. Stärken

Unser Projekt ist übersichtlich umgesetzt und gut strukturiert. Wir setzen auf neue Rails 3 Features und haben wiederverwendbaren Quellcode geschrieben. Es werden Techniken wie Vererbung eingesetzt um den Code nicht kopieren zu müssen.
Die Test-Abdeckung ist unserer Meinung nach genau richtig für ein Projekt dieses Umfangs.

## 3. Schwächen

Unser Datenbank-Schema ist für ein dokumentenbasiertes Datenbank-System wie Mongo ungeeignet und besser für den Einsatz in einer relationalen Datenbank. Dadurch haben wir eine große Datenmenge und lange Abfragezeiten. Teilweise sind die Abfragezeiten so lange, dass wir sie gar nicht zu Ende laufen lassen konnten.

## 4. Verbesserungsvorschläge

Datenbankstruktur:
Das Projekt wäre mit einer anderen Datenbank-Struktur wie beispielsweise embedded Documents oder alles in jeweils einem Dokument besser strukturiert. Beziehungen zwischen einzelnen Dokumenten sollten unbedingt vermieden werden.

Datenbank-Verteilung:
Vielleicht wäre unser Projekt mit einem verteilten Datenbanksystem performanter. Wir haben hierbei an den Einsatz von mehreren MongoDB-Instanzen parallel oder Themen wie Sharding oder Ähnliches gedacht.

## 5. Fazit

Wir haben Erfahrungen gesammelt und uns weiterentwickelt sind allerdings an der großen Datenmenge und dem falschen konzeptionellen Ansatz gescheitert. Wir können dies aber frei zugeben, da wir nun um diese Erfahrung reicher geworden sind.

Das gesamte Projekt ist zu finden unter:
*https://github.com/lpsBetty/alarmproject*