EBI Documentation

Release 0.1.0-alpha

Antoine Kalmbach

January 08, 2016

Contents

A modern application architecture

This repository contains a description and an example implementation examples of the **Entity—Boundary—Interactor** (**EBI**) application architecture, derived from ideas initially conceived by Uncle Bob in his series of talks titled Architecture: The Lost Years and his book.

The EBI architecture is a modern application architecture suited for a wide range of application styles. It is especially suitable for web application APIS, but the idea of EBI is to produce an implementation agnostic architecture, it is not tied to a specific platform, application, language or framework. It is a way to design programs, not a library.

The name **Entity–Boundary—Interactor** originates from a master's thesis where this architecture is studied in depth. Names that are common or synonymous are **EBC** where *C* stands for **Controller**.

Examples of how to implement the architecture are given in this document and are written in *Elixir*, a dynamically typed language with a simple and powerful syntax.

> **Warning:** This is still very much a work in progress. Contributions are welcome on Github.

CHAPTER 1

Introduction

> The architecture of something *screams* the intent.
>
> —Robert C. Martin

Often when looking at the code of web applications, you are greeted with a mass of folders, library installation, and tooling configurations. The code is structured haphazardly into nondescriptive folders like `app`, and the dependency layout of the application is a jungle.

As a result, the purpose and architecture of the program become opaque.

> The architecture of an application is driven by its use cases.
>
> —Ivar Jacobsen

The idea is to design programs so that their architectures immediately present their use case. It's a way to design programs so that its internal dependency graph is organized cleanly and its elements are joined together with as loose coupling as possible.

Ultimately, the goal is the *separation of concerns* between application layers, this architecture and many like it aren't dependent on presentation models or platforms. All the arrows, or dependencies, point inwards in the abstraction chain, each successive layer less abstract than the one before it.

Keeping the arrows pointing inwards makes the code easy to maintain, extend, test and refactor. EBI imposes some architectural requirements on the programmers, that is, you must navigate around its rules, but this is kept at a minimum.

1.1 How does this archtecture differ from MVC?

The difference between EBI and MVC is that an EBI architecture is that the business logic of the application is designed to be platform-agnostic of its delivery mechanism.

To paraphrase, this means that the business logic parts, interactors and entities, do not know in which medium they're being accessed from. It could be from a web server, a unit test, or a GUI application.

Contrast this to MVC, where there is *always* a dependency on the delivery mechanism. No matter how hard you tried, you cannot tear Rails controllers out of the web world.

What makes decoupling the business logic from the delivery mechanism a good thing? This is outlined in the next section, **'ref':design:**.

CHAPTER 2

Design Goals

Todo

write summary

2.1 Extensibility

2.2 Testability

2.3 Stability

2.4 Summary

The above architecture is suited for any language and **any use case**. One only needs an ability to define abstractions, were they type classes, interfaces, OCaml modules, Rust traits, or Clojure protocols. Static typing is not required here: you just need *one* way of creating clear and verifiable functionality definitions. In dynamically typed languages like Clojure and Elixir you can use protocols (with runtime assertions), or even just plain old documentation. The boundary layer needs only to be *specified*, it's not a strict language requirement.

The arrows in this architecture tend to point inwards. Only the middle layer (the service layer) is seen by both the Core and the API layer is because it describes the language of the system, but none of its functionality.

Keeping the arrows unidirectional will make the system more robust and scalable. If you decide to port your GUI app to a web service the interactors will stay the same.

Moreover, unit testing is easy: you can mock *anything*, and what is more, the unit tests will be fast and simple. Entities will only test their internal business logic, interactors will not fumble with web services, the presentation layer will only deal with handling requests and responses and calling the right interactor, the host layer will contain system-specific tests (e.g. HTTP tests), but **all** of these components can be tested separately in a horizontal fashion.

CHAPTER 3

The Architecture

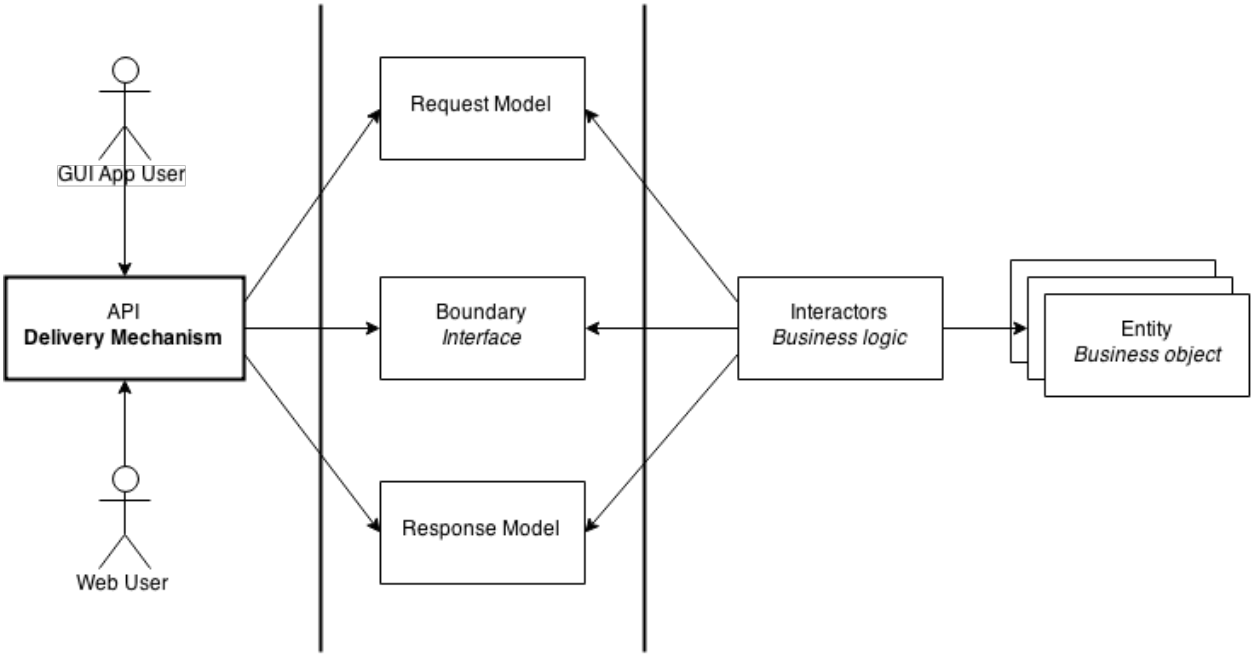


Fig. 3.1: An overview of all the logical units of the architecture.

The architecture can be approached from two different perspectives. The first is the dependency graph, as you can see above. The second is the hierarchy graph, which presents a concrete separation in a program.

The architecture is best described as a *functional data-driven* architecture, where requests are processed into results. The architecture consists of three different components.

- **Entities** are the core of the architecture. Entities represent business objects that have application independent business rules. They could be `Books` in a library or `Employee` in an employee registry. All the application agnostic business rules should be located in the entities.
- **Boundaries** are the link to the outside world. A boundary can implement functionality for processing data for a graphical user interface or a web API. Boundaries are functional in nature: they accept data *requests* and produce *responses* as result. These abstractions are concretely implemented by interactors.
- **Interactors** manipulate entities. Their job is to accept requests through the boundaries and manipulate application state. Interactors is the business logic layer of the application: interactors *act* on requests and decide what to do with them. Interactors know of request and response models called **DTOs**, data transfer objects. Interactors are **concrete** implementations of boundaries.

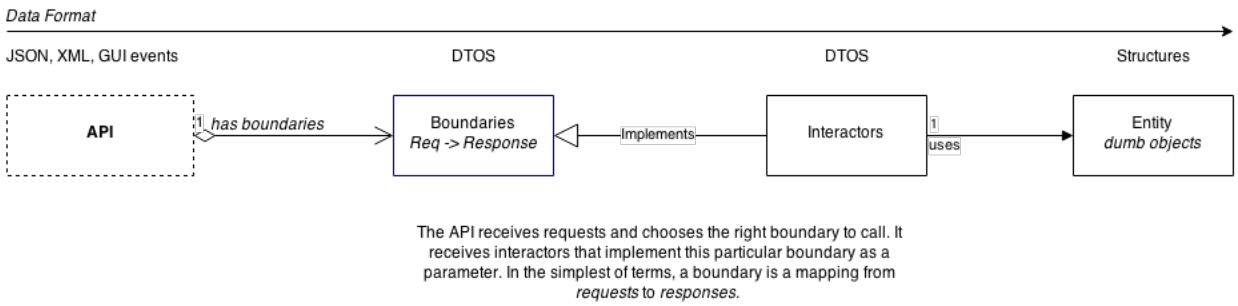


Fig. 3.2: An object diagram of the program.

CHAPTER 4

Structuring Applications

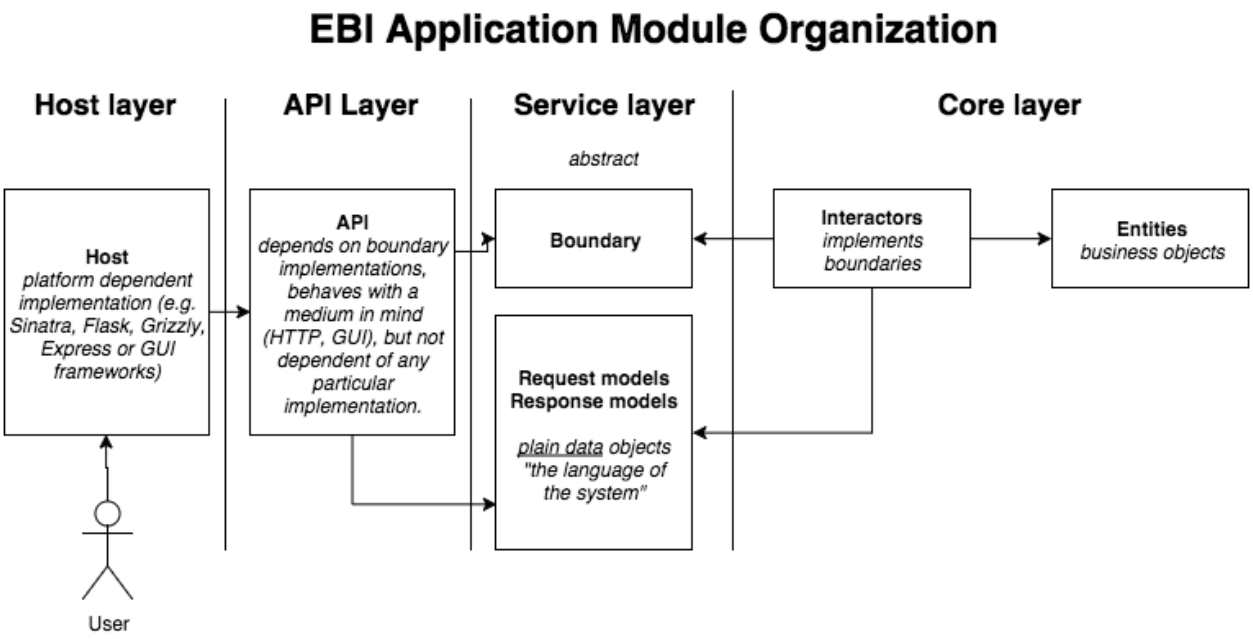


Fig. 4.1: The code-level organization of modules. Each vertical section is an separate module of the program.

Furthermore, it is good practice to separate the EBI architecture itself into five different layers. These layers correspond to namespaces or packages in your language of choice.

- The **Host** layer implements a physical manifestation of the API, e.g., a web server
- The **API** layer is the interface to the program itself, which accepts input and translates it into DTOs, passing them to
- The **Service** layer that contains **boundaries** and **response** and **request** models
- The **Core** layer that contains a concrete implementation of the service layer
- **Interactors** which implement boundaries and form the core business logic of the application
- **Entities** which represent the data models of the program

Thus, when a program is constructed, the API is built top-down using dependency injection. The **Host** layer is the one doing the DI of the concrete interactors.

And that's it. The interactors do not know what protocol its requests come from or are sent to, and the API doesn't know what sort of an interactor implements the service boundary.

CHAPTER 5

Event Lifecycles

Todo

write summary

5.1 Web: the life-cycle of a HTTP request

A *request DTO* enters the application via the request boundary. This is usually the API layer sitting on top of some interactor. In the pictured example, we have a `GetGopher` interactor whose task is to retrieve information about a store of gophers, accepting `GopherRequest`s and returning `GopherResponse`s. The *user interaction* is the request DTO and in this example is in plain JSON.

The interactor `GetGopher` then can be seen as a mapping of `GetGopherRequest`s to `GopherResponse`s. Because the requests and responses are **plain dumb objects**, this implementation is not dependent of any technology. It is the duty of the API layer to translate the request from, e.g., JSON, to the request DTO, but the interactor doesn't know anything about the protocol or its environment.

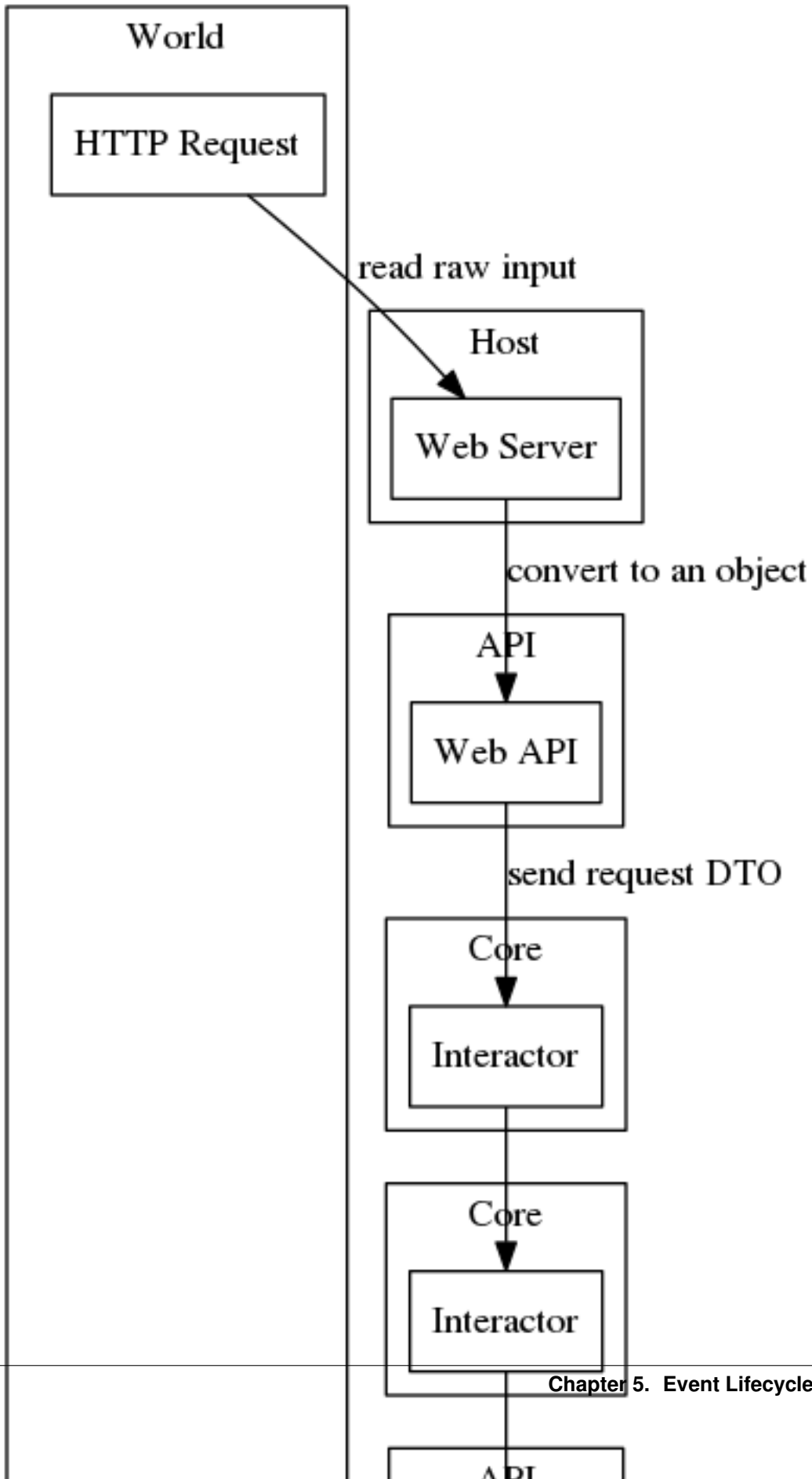
5.2 GUI: the life-cycle of an event

In the GUI model, the architecture looks a bit simpler, since the host layer does all the important work for us. We don't need to worry about serialization so much anymore.

The API layer of a GUI application translate user interactions to request objects. The Host layer sends an `AddAuthorButtonClick` event to the API. The API also listens to the `AuthorNameFieldEdited` event using which it keeps track of the contents of the form field. Once the `AddAuthorButtonClick` event is received, the API creates a `CreateAuthor` request DTO with the appropriate details and sends it to the interactor for processing. Once the interactor returns, the API layer pushes a response to the Host via some mechanism, e.g., by formating and sending a `UserAdded` event to the Host layer, which in turn handles the updating of the user list component using its own logic.

As you can see, a GUI application implemented with EBI is *instantly* a lot more complicated than a simple web server. This isn't a coincidence, it's natural: GUI apps **are** really complicated underneath.

The extra cost of such abstractions is that as the interactor and entity layer remain unmodified, you can easily swap the API and Host layer for another implementation. So you're never tied to any certain interaction or delivery mechanism.


World
HTTP Request
read raw input
Host
Web Server
convert to an object
API
Web API
send request DTO
Core
Interactor
Core
Interactor

CHAPTER 6

Overview

From the directory tree one can see that the code is organized into separate namespaces. In the example implementation this is achieved by splitting the code into different folders, since this is a one-to-one mapping to packages (namespaces) in the Go programming language.

The `api` folder contains the API, the `host` web servers or GUI apps, the `service` contains the boundary layer with the request and responses models, the `core` layer contains the core program architecture hidden from view.

As mentioned previously, the purpose of the program should be visible by looking at it. By exploring the `service` directory (containing `gophers.go` *et al.*) we can immediately see the services this program provides.

CHAPTER 7

The Presentation Layer

The introduction of the API layer at this point may seem a bit heavy-handed. Why not map the interactor methods directly to routes? I mean, you could spin up a web server that handles Sinatra-like requests and then points them to the right interactor and returns a serialized version of whatever the interactor spewed out.

Indeed, if you have *one* interactor or a couple that don't ever do business together, this seems like the right approach. Once you get too many, it gets useful to wrap them underneath a single unit.

Suppose you have an API endpoint of a book catalogue, and you want to implement functionality that that modifies a particular author and at the same time transfers these modifications to the publications. You receive the new author name as input, and then you must update the author itself and their book catalogue in one go.

Sure, sounds easy, just create a `ModifyCatalogue` functionality into the `Author` interactor. The interactor, in this case, would modify the author's name, then loop over its `Books` and modify them individually, finally sending the updates to a database. This system works as long as the `Book` entities are under the sole ownership of an `Author`–that is, there is no way of adding, creating, deleting, or modifying a book from outside.

As soon as you introduce a `Book` interactor into the mix, things start to get hairy. The `Author` service, retaining its book modification logic, now overlaps with the `Book` service. The imminent solution to this is to lift this logic from the `Author` interactor to the `Book` interactor, making the layout look like this.

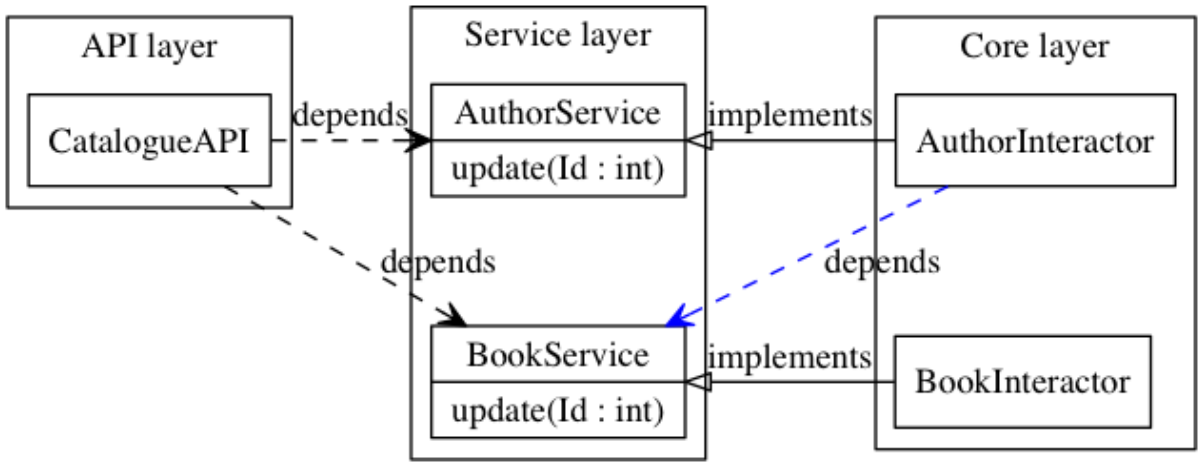


Fig. 7.1: Could the blue arrow be removed, and contained inside the arrows from the API layer pointing towards the Service layer?

The blue dashed arrow can be lifted into the API layer with little extra work. It's a good idea to push such arrows as far

"up" as possible, because this helps keep one thing in check: not violating the **single responsibility principle**, which roughly means that your interactor should do one thing and **one thing only**. So the Author interactor should only care about author logic, and the Book interactor should care only about book issues.

In the above example this process would not be violated if there was no Book service, such that book-related logic was underneath the Author interactor. But, as soon as you start sharing responsibilities, and they start to overlap, you will run into problems.

Hence, the API layer is there to provide additional logic that ties two interactors together. You could think of it as a *meta-interactor*, something that operates on interactors only, but contains no low-level business logic.

What is more, the API layer usually has some knowledge of the application domain: while interactors deal with dumb objects (DTOs), the API may be dealing with HTTP request objects. Thus, the API is closer to the actual implementation.

Consequently, the **Core** layer is the non-duplicated, non-overlapping part of the application: you may have multiple APIs for the same set of interactors, and multiple *hosts* for each API, but at the fundamental level, there's only one canonical implementation of the core.

To conclude, the key differences between an API and an interactor are the following:

- An API is domain-specific and knows about the target implementation. The API knows it is talking to a web server. It just doesn't know *which kind* of web server it is talking to, acting as a bridge between interactors and the delivery mechanism.
- The API layer may tie a multitude of interactors together, without making them dependent on each other, enforcing loose coupling.
- APIs can be seen as "meta-interactors", operating on interactors the same way interactors operate on entities.

CHAPTER 8

The Boundary Layer

Todo

flesh out

8.1 Designing good DTOs

DTOs have no business logic. Think of them as language constructs around simple requests not dependent of any protocol.

In our Go program, the naming convention is to have a service "Foobar" (in caps, can be a pluralized noun), and have it in `service/foobar.go`, and its request and response models are *all* in `service/requests/foobar.go` and `service/responses/foobar.go`.

Though these interfaces are named similarly, in Go, we refer to these types as `requests.FindGopher`, hence it is never ambiguous as to what the structures are. The `requests` (or responses) packages contain only structures like these, hence there will never be any confusion between the two.

In other languages, you would usually have a suffix of some sorts or use a namespace explicitly to avoid repetition.

8.2 Wrapping up

The service layer is the common language of the application architecture. When the API and core speak to each other, they do so via an abstract boundary. They use DTOs (data transfer objects), simple structures of data, for communication. We now move on to the core layer of the architecture.

CHAPTER 9

The Core Layer

The core layer contains actual business logic. First we start off with the entity, the rich business objects of the application. In `core/entities/entity.go`,

9.1 Entities

```
type Gopher struct {
    Name string
    Age int
}
```

Entities are completely invisible to the outside layers. Not any thing but the interactors know about them. Entities contain business logic, e.g., a `Gopher` entity can modify itself or contain functions related to it, but the distinction between entities and interactors is the following:

- entities modify *themselves* vs.
- interactors modify *entities*

An entity can contain other entities: a `Gopher`, could technically possess a `Tail` and two `Eyes`, and it can modify them at will. This hierarchy is strictly unidirectional: a `Gopher` doesn't know about other gophers, more importantly, *it doesn't know about the interactor*.

9.2 Interactors

Interactors contain rich business logic. They can manipulate entities and they implement boundaries. Here, we have the `Gophers` boundary from above to implement, so we implement a smallish interactor that implements it.

```
type Gophers struct {
    burrow map[int]entities.Gopher
}

func NewGophers() *Gophers {
    return &Gophers{
        burrow: make(map[int]entities.Gopher),
    }
}
```

It implements the three methods as defined by the `Gophers` boundary

```
// Find finds a gopher from storage.
func (g Gophers) Find(req requests.FindGopher) (responses.FindGopher, error) {
    gopher, exists := g.burrow[req.ID]
    if !exists {
        return responses.FindGopher{}, errors.New("Not found.")
    }

    return gopher.ToFindGopher()
}

func (g Gophers) FindAll(req requests.FindGopher) ([]responses.FindGopher, error) {
    var resps []responses.FindGopher
    for _, gopher := range g.burrow {
        fg, err := gopher.ToFindGopher()
        if err != nil {
            return []responses.FindGopher{}, err
        }
        resps = append(resps, fg)
    }
    return resps, nil
}

// Create creates a gopher.
func (g Gophers) Create(req requests.CreateGopher) (responses.CreateGopher, error) {
    var gopher entities.Gopher
    if err := gopher.Validate(req); err != nil {
        return responses.CreateGopher{}, err
    }

    gopher.ID = g.getFreeKey()
    gopher.Name = req.Name
    gopher.Age = req.Age
    g.burrow[gopher.ID] = gopher

    return responses.CreateGopher{ID: gopher.ID}, nil
}
```

As one can see, the interactor is completely unaware of any protocol dependencies. The relation to web applications is obvious: we are, after all, talking about requests and responses, and the DTOs translate very easily to JSON objects. But they can be used without JSON, in fact, the whole point is that even a GUI application will pass the same objects around.

The interactors (and by extension, entities) are completely oblivious to their environment: they don't care whether they are running inside a GUI application, a system-level daemon, or a web server.

9.3 Beware of Behemoths

Interactors are business logic units. How much business logic is too much business logic? The best rule of thumb is the **single responsibility principle**: an interactor should only do one thing, and one thing only. I'm also going to address this *below*, but the most important thing to understand about interactors is that they should operate only one *one* aspect of the business logic.

What this means may not be immediately clear. If you are building a REST API, you will generally have some separation of concerns already going on at the external API level, in the form of URIs. To use a book catalogue as an ad hoc example, you could have a URI for book authors at `/authors` and `/books`, these clearly indicate—to the API user, anyway—what lies beneath.

At the code level, this distinction must be maintained. An author may contain a collection of books they have, but whose responsibility is modifying them? Obviously, since we have two URIs here, one for books, one for authors, we must decide which one handles the logic of modifying book entities. In this case, any internal *modification logic* of the book entities must reside underneath a **single** interactor. There can be two cases here:

- **One interactor does everything**. The `/books` URI is just an alias underneath the Author interactor, or vice versa.
  - **Pros**: no overlap in logic, no conflicts, since everything is contained under one unit (a single interactor).
  - **Cons**: must be split eventually, since otherwise it will grow to monstrous proportions.
- **Two interactors, ‘‘AuthorInteractor‘‘ and ‘‘BookInteractor‘‘**. The `AuthorInteractor` calls methods of the `IBookService` (which `BookInteractor` implements) to modify the `Book` entities contained (or *owned*) by an `Author` entity.
  - **Pros**: no chance of overlap since the responsibilities are split.
  - **Cons**: risk of introducing circular dependencies between boundaries (see *below*).

If you’re building a really simple service, you don’t *have* to split interactor duties, but it’s a good idea. Be careful of choosing short-term practicality in favor of long-term abstractions, it may bite you in the rear one day!

As a summary, in the presented example, the `AuthorInteractor` should only modify things related to `Author`s, and preferably only *read* data about `Book`s, leaving modification and updates to the `BookInteractor`. There are two ways on how to implement the necessary communication, that is, how the `AuthorInteractor` calls the `BookInteractor`, and this will be resolved later, but now we have a small interlude about something equally vital: the external world.

CHAPTER 10

Talking to the External World

One part I haven't yet addressed in this overview is how to talk to external dependencies, like a database. The answer is remarkably simple: create them behind a boundary and build them like an interactor. This enforces loose coupling, and the interactors *still* talk to each other using interfaces.

Similarly, if you're building a GUI application and want to use events, the interactor can push events to an event broker boundary, or the API layer can handle the responses from the interactor, and call other interactors through their boundary interfaces. This brings us to the API layer.

CHAPTER 11

Example Implementation

A REST API in Elixir

This document features a small example implementation of this architecture in *Elixir*. Elixir is a dynamically typed language leveraging the Erlang virtual machine.

I chose Elixir because of its simple but powerful syntax. I originally wanted to implement this in Ruby but I wanted clear examples of *interfaces* and Ruby doesn't really have them. Thankfully, Elixir has *protocols*, which let me write the boundary descriptions using a high-level abstraction.

Note: Interfaces aren't absolutely necessary.

You don't really *need* interfaces to implement boundaries, the language-level abstractions make it easier to understand in its own terms. Since no boundary object is an actual, concrete implementation, it quickly becomes obvious that the boundary objects, and thus the service layer, act as a *data model* inside the system.

For Ruby and Python you could easily write a dummy abstract class with `NoMethodImplementation` exceptions being thrown left and right, in case of an unsatisfied boundary.

The Elixir implementation makes the use of the Spirit microframework for Elixir. Equivalent frameworks in other applications:

- **Ruby**: Sinatra, Cuba
- **JavaScript**: Express
- **Go**: net/http
- **C#**: ServiceStack
- **Java**: SparkJava

...and so on.

```
.
-- api
|   -- web_api.ex
-- entities
|   -- author.ex
|   -- publication.ex
-- host
|   -- server.ex
-- interactors
|   -- author_service.ex
|   -- publication_service.ex
-- service
```

```
    -- protocols.ex
    -- requests.ex
    -- responses.ex
```

CHAPTER 12

License

CHAPTER 13

FAQ

faq off