IDEAS FOR A CLEAN ARCHITECTURE WITH QT

HELLO!

I am Daniele Maccioni

Architectures are fun!

- gendoikari@develer.com
- 🐦 @gendoikari_nerv
- https://github.com/GendoIkari

```
ef="index.html">Home</a></li>
ef="home-events.html">Home Events</
ef="multi-col-menu.html">Multiple C
="has-children"> <a href="#" class=
li><a href="tall-button-header.htm
li><a href="image-logo.html">Imag
li class="active"><a href="tall-
>

s="has-children"> <a href="#">Ca

<li><a href="variable-width-slic
<li><a href="testimonial-slider
<li><a href="featured-work-slic
<li><a href="equal-column-slid
<li><a href="video-slider.html
<li><a href="mini-bootstrap-c
>
```

“It is not enough for code to work.”

Robert C. Martin

THE "PROBLEM"

A conversation about a "classic" architecture

1

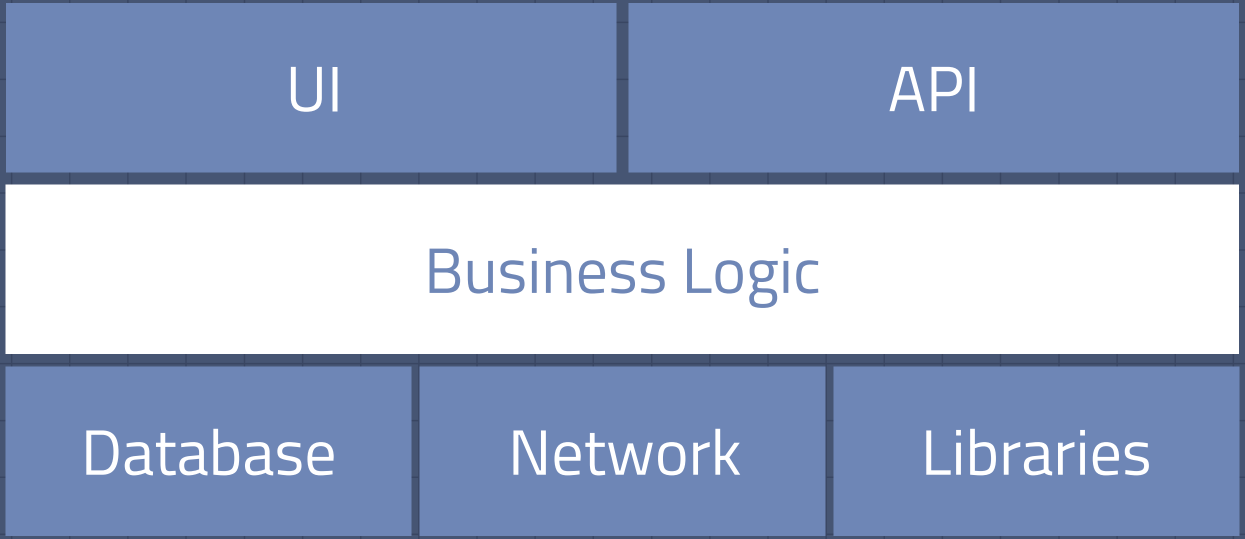
WHAT'S A SOFTWARE ARCHITECTURE?

- How functions, classes and modules work together
- The strategy behind the structural choices
- The big picture, the vision, the evolution

Architecture is NOT about frameworks, tools, libraries.

Architecture is about the use-cases of the problem you are trying to solve.

SIMPLE BUT EFFECTIVE IDEA



A simple architecture is infinitely better than no architecture.

PROS & CONS

Pros

- Simple
- Easy and fast to implement
- Relatively easy to maintain
- Easy to teach
- It just works

Cons

- Easy to make a mess
- The big picture is very low resolution
- Business logic depends on implementation details
- Easy to end up with everything depending on everything else

IMPROVEMENTS

Let’s talk about few ideas

2

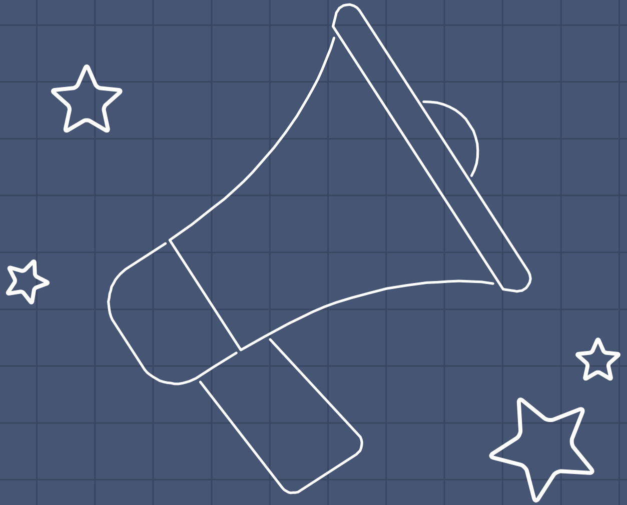
EXAMPLE: RSS READER

- Simple case to sketch few ideas for our architecture
- Overengineering

```
<rss>
     ...
     <item>
          <title>My Article</title>
          <description>That's a description.</description>
          <link>https://...</link>
          <pubDate>Fri, 18 May 2018 11:00:00 GMT</pubDate>
     </item>
     ...
</rss>
```

SCREAMING Architecture

The plan should scream its purpose to the world

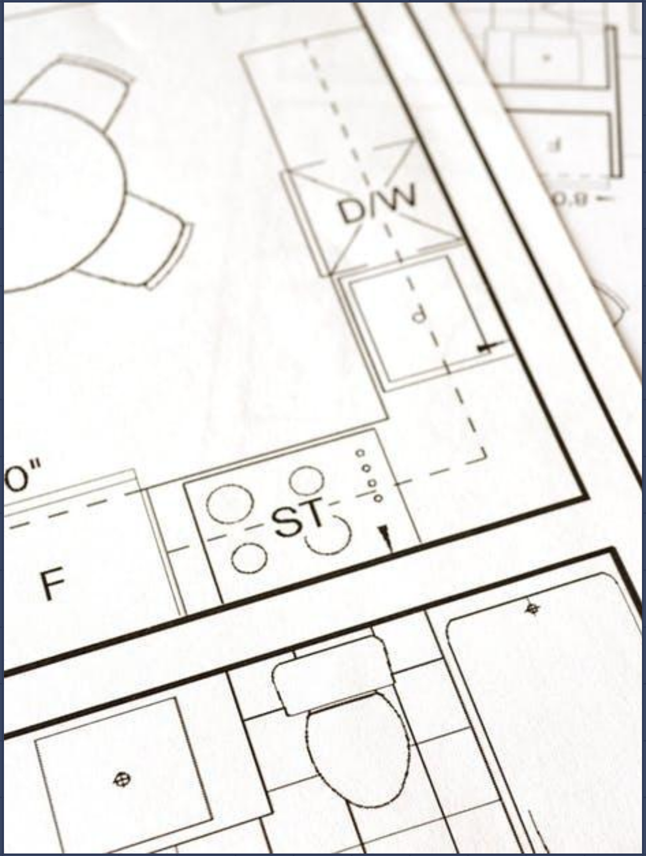


Building Metaphor

Irrelevant: walls materials, colors, how many workers needed, what tools used…

Relevant: function, features, what rooms, their functions…

You look at the plan and you clearly see the purpose of the building.



THE DOMAIN

Feed

A list of articles downloaded from an online newspaper.

Channel, title, description, language, entries…

Entry

A single article contained in a feed.

Title, author, description, link, pubDate…

Personas

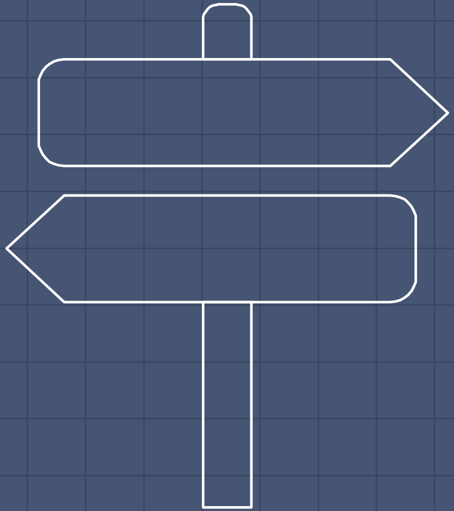
The list of the archetypical users of our system: in our case a reader.

WHAT'S MISSING?

- Framework, database, protocols...
- Server? Web? FileSystem?
- Models? Views? Controllers?

These are all details!

PROCRASTINATE BIG DECISIONS



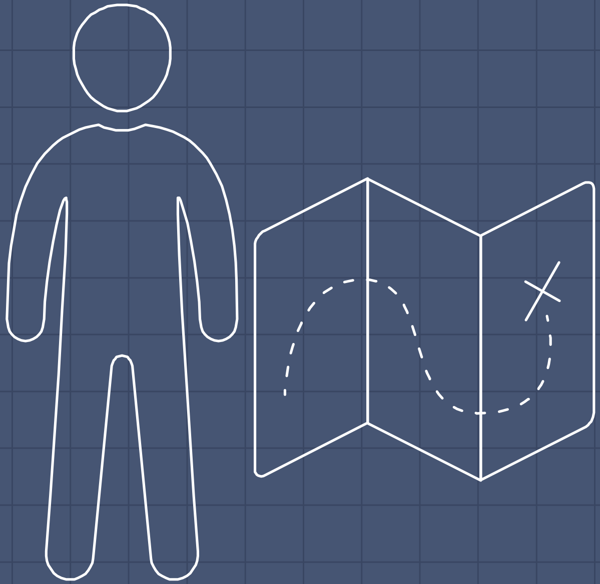
A good architecture should allow you to wait to commit yourself to an unclear decision

> “A good architecture maximizes the number of decisions NOT made.
>
> Why? Because later is always better when you make a decision: you have more information.”

Robert C. Martin

USE CASES

The center of your application is what it can do



USE CASES

Load entries from default rss xml

Description: the entries from the default feed are downloaded and shown.
Trigger: the application startup.

Flow: the default url is queried at startup to download the last entries from the feed. The xml is parsed and the articles are shown in the main page.

Set an url as default feed

Description: the default feed is changed by the user.

Trigger: …

Flow: the url given by the user is saved as default feed.

GOING CLEAN

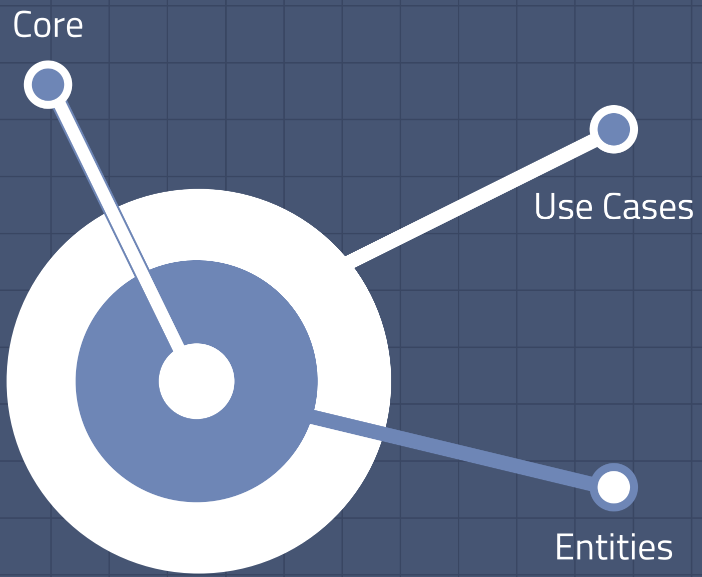
3

STARTING A NEW DIAGRAM…

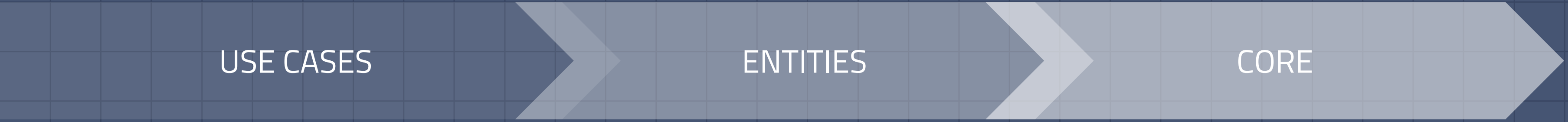
- The **core** objects are the center of the architecture. *std::string, std::vector, QString, QVector…*

- The **entities** are the smart objects of our **domain**. *Feed, Entry, User…*

- Use Cases use entities to implement changes and actions in our **business logic**.



DEPENDENCY FLOW



USE CASES

ENTITIES

CORE

The outer elements depend on the inner elements.

Entities know and use core elements and know NOTHING about use cases.

Here we have only basic structures, containers, standard libraries...

Entities for the Business Logic

```
class Feed {
public:
   static Feed* fromXML(QByteArray xml);
   Entry* addEntry(EntryData entryData);
   void removeEntry(int entryID);
   Entry* entry(int entryID);
   QVector<Entry *> entries();
   QVector<Entry *> filteredEntries(FilterData filter);
   ...
private:
   ...
};
```

```
class Entry {
public:
   Entry(EntryData data);
   EntryData toData();
   QString title();
   QString author();
   QUrl url();
   bool isAlreadySeen(QDateTime time);
   ...
private:
   ...
};
```

(LET'S NOT FORGET ABOUT S.O.L.I.D.)

Single-responsibility

A class (entity) should have only one job.
Keep responsibilities separated in different "atomic" objects.

Liskov substitution

Every derived class should be a working substitutable for their base class.

Use Cases

```
class LoadDefaultFeed {
public:
   ...
   void execute(LoadRequest request, EntriesPresenter& presenter);
   ...
};
```

```
struct LoadRequest {
   QUrl feedURL;
   ...
};
```

```
class EntriesPresenter {
public:
   virtual ~EntriesPresenter() = default;
   virtual void presentEntries(EntriesResponse response) = 0;
};
```

(LET'S NOT FORGET ABOUT S.O.L.I.D.) part 2

Open-closed principle

Objects should be open for extension but closed for modification.

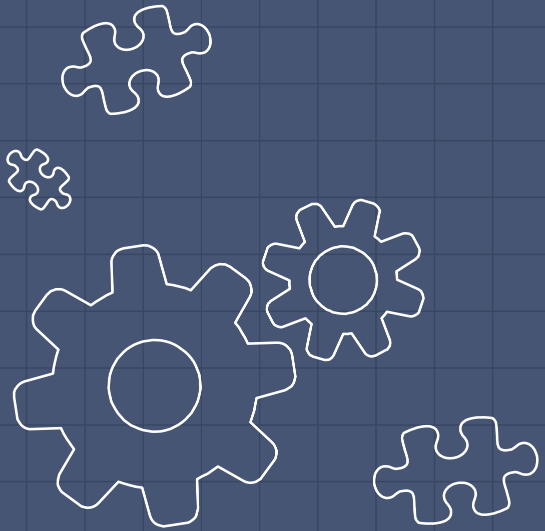
Interface segregation

Single responsibility principle for interfaces: a client should never be forced to implement an interface that it doesn't use.

Dependency Inversion

Objects should depend on abstractions not on concretions.

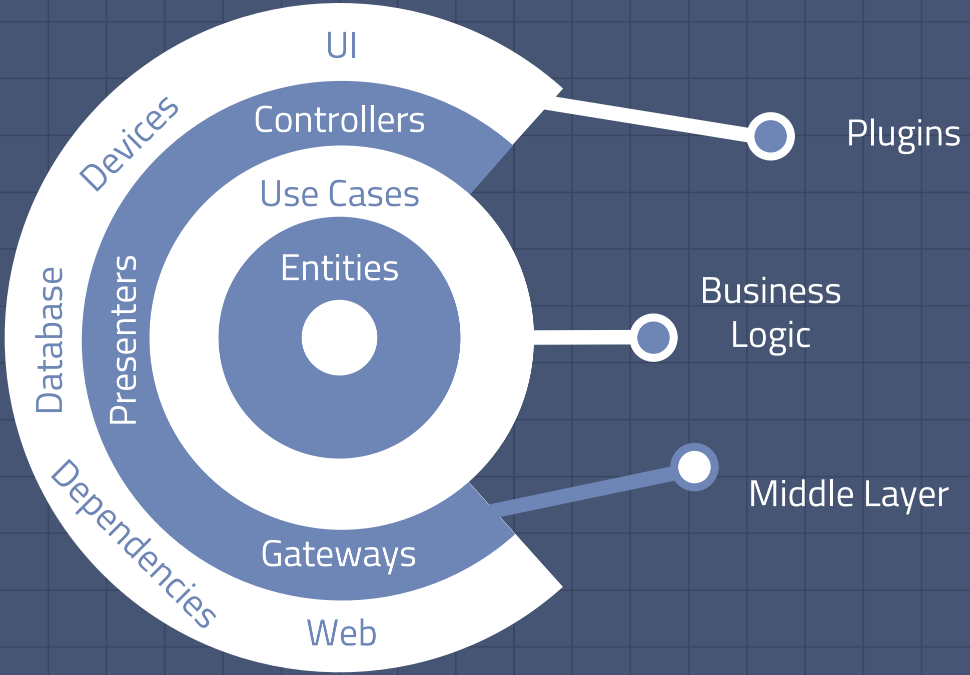
PLUGINS



Plugins depend on your application, not the other way around.

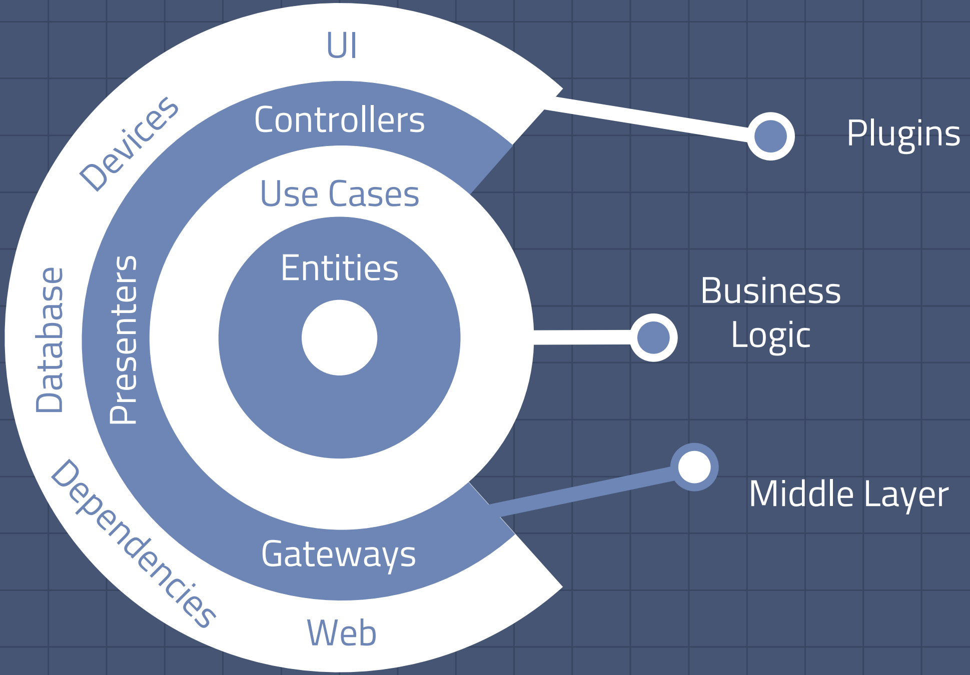
EVERYTHING IS A PLUGIN

- **Controllers, presenters and gateways** depend on the inner business logic.
- **Gateways** are the doors to the external world. Your application get data from gateways and send data to gateways.
- **Presenters** and **controllers** *handle* inputs and outputs.
- Everything else (*databases, ui, external devices, web…*) is a **plugin!**



WHERE'S QT?

- It depends on how much we want to be tied to Qt/QtQuick
- Qt can be in the **Core**, using Qt containers, QFuture…
- Qt can be in the middle layer, with controllers deriving from QObject
- For sure in the UI
- It probably shouldn't be in the business logic



Gateways

```
class FeedRepository {
public:
   virtual ~FeedRepository() = default;
   virtual QFuture<Feed*> feed(int feedID) = 0;
   virtual void saveFeed(Feed* feed) = 0;
   virtual QFuture<Feed*> retrieveFeed(QUrl url) = 0;
   ...
};
```

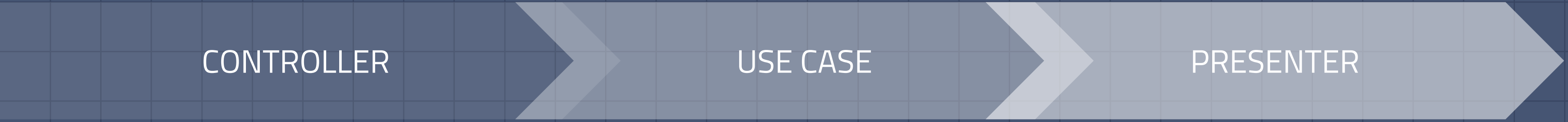
```
class WebFeedRepository {
public:
   QFuture<Feed*> feed(int feedID);
   void saveFeed(Feed* feed);
   QFuture<Feed*> retrieveFeed(QUrl url);
   ...
private:
   QNetworkAccessManager manager;
};
```

The use case implementation

```
void LoadFeed::execute(LoadRequest request, EntriesPresenter& presenter)
{
    auto futureFeed = Context.feedRepository->retrieveFeed(request.feedURL);

    whenFinished<Feed*>(futureFeed, [&](Feed* feed) {

        QVector<EntryData> entriesToShow;
        for (auto entry : feed->entries()) {
            entriesToShow.append(entry->toData());
        }

        presenter.presentEntries(EntriesResponse{
            entriesToShow,
        });
    });
}
```

```
struct context {
    std::unique_ptr<FeedRepository> feedRepository;
};
```

ACTION LIFETIME



CONTROLLER

USE CASE

PRESENTER

The external UI layer uses controllers to send actions and data to inner circles.
Here we have QObject, connects, signals and slots.
The controller produces requests.

Use cases do all the dirty jobs.
They use entities and the functions provided by the business logic.
Use cases produce response for the presenters.

The presenter handles the response from the business logic and provides ui-friendly data to be shown in gui.

Controllers

```
class MainController : public QObject {
   Q_OBJECT
   Q_PROPERTY(QList<QObject*> entries READ entries NOTIFY entriesChanged)
   ...
public:
   ...
public slots:
   void loadEntriesFrom(QUrl url);

signals:
   void entriesChanged();


   ...
};
```

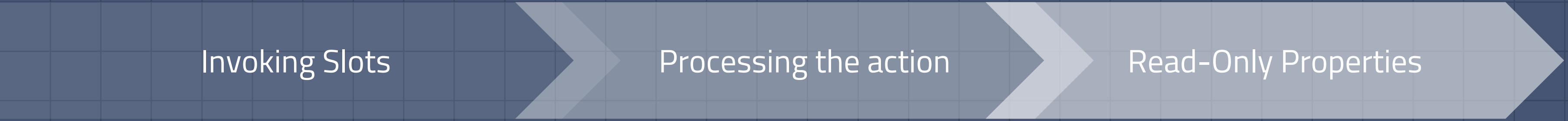
```
rss::Context.feedRepository =
                    std::make_unique<rss::WebFeedRepository>();


  qmlRegisterType<ui::MainController>("RssFeed", 1, 0, "Main");
  qmlRegisterType<ui::EntryController>("RssFeed", 1, 0, "Entry");
```

The Actual UI

```
  Main {
     id: mainController
  }

GridView {
  model: mainController.entries
  delegate: EntryDelegate {
     ...
  }
}
```

```
Component.onCompleted: {
   mainController.loadEntriesFrom("https://www...../rss")
}
```

DATA FLOW



Invoking Slots

Processing the action

Read-Only Properties

From QtQuick we are calling slots in the controllers below to send actions (a little bit like redux).

The use case do what it must do, activating the presenter at the end. The presenter process the data.

The presenter sends updates toward the UI (QtQuick again) as read-only models and read-only properties.

Implementing Presenter Interface

```
class MainController : public QObject, EntriesPresenter  {
   Q_OBJECT
   Q_PROPERTY(QList<QObject*> entries READ entries NOTIFY entriesChanged)
public:
   …
   void presentEntries(rss::EntriesResponse response);

public slots:
   void loadEntriesFrom(QUrl url);

signals:
   void entriesChanged();
   ...
};
```

```
void MainController::loadEntriesFrom(QUrl url)
{
   rss::LoadFeed().execute(rss::EntriesRequest{ url }, *this);
}
```

IS IT WORTH IT?

It Helps A Lot

- The single pieces are very well defined
- The dataflow is very well defined
- Dependencies fixed
- Almost everything is extendible

- Use cases at the very center
- Business logic isolated
- Application logic possibly reusable
- You can test everything
- Mocking is easy
- UI becomes a very very thin layer

THANKS!

Any questions?

You can find me at

- gendoikari@develer.com
- https://github.com/GendoIkari