Arquitectura de l'Aplicació Flutter

v.0.1

Índex

Objectiu del document

Aquest document descriu detalladament l'arquitectura tècnica que conforma l'aplicació Flutter pel projecte Sabina.

Sempre que sigui possible s'evitarà l'ús de llibreries de tercers.

Per a la comunicació asíncrona entre els diferents elements de l'aplicació es faran servir Stream's natius de Dart.

Classes del nucli de l'arquitectura

Mixin: *LdTagMixin*

```
mixin LdTagMixin {
  // 🧩 MEMBRES ————————————————————
  bool _isTagSetted = false;
  late final String _tag;

  // 🕹️ GETTERS/SETTERS ————————————————
  String get tag ⇒ _tag;
  set tag(String pTag) {
    assert(!_isTagSetted && pTag.isNotEmpty);
    _isTagSetted = true;
    _tag         = pTag;
  }
}
```

Totes les instàncies de classes que hagin d'actualitzar-se degut a qualsevol tipus d'event asíncron han d'incorporar aquest *mixin*. D'aquesta forma les instàncies contindran una cadena *tag* identificadora que permetrà identificar tant a la font de l'event com als *tags* destí (sempre que l'objectiu no sigui enviar un event a tots els oïdors, on el paràmetre de destí quedaria nul).

El tag d'una instància no és modificable ni pot ser nul.

Interfície: *LdTagIntf*

```
abstract class LdTagIntf {
  /// Retorna el nom base de la classe per a utilitzar com a prefix
  /// de tag quan aquest no s'especifica a partir dels constructors.
  String get baseTag;
}
```

Aquesta classe abstracte fa les funcions d'interfície. El seu objectiu és forçar la implementació del '*get baseTag*' en totes les classes on haguem de disposar d'una base estàndard per a la generació dels *tags* identificadors.

Classe: *LdBinding*

```
class LdBinding<T extends LdTagMixin> {
  // 📝 ESTÀTICS ————————————————————
  static int _views = 0;
  static String get _newView ⇒ NumberFormat('00000').format(_views++);
  static int _widgets = 0;
  static String get _newWidget ⇒ NumberFormat('00000').format(_widgets++);
  static int _states = 0;
  static String get _newState ⇒ NumberFormat('00000').format(_states++);
  static int _ctrls = 0;
  static String get _newCtrl ⇒ NumberFormat('00000').format(_ctrls++);

  // 📝 SINGLETON ———————————————————
  static final LdBinding _single = LdBinding._();
  static LdBinding get single ⇒ _single;

  // 🧩 MEMBRES ————————————————————
  final LdMap<T> _map = LdMap<T>({});

  // 🛠️ CONSTRUCTORS ———————————————
  LdBinding._();

  // 🌥️ GESTIÓ DELS BINDINGS  ——————
  // Afegeix un objecte al magatzem.
  void add(String pTag, T pBind) {
    _map[pTag] = pBind;
  }

  // Elimina un objecte del magatzem.
  LdTagMixin? remove(String pTag) {
    return _map.remove(pTag);
  }

  // Retorna cert només si la clau està enregistrada.
  bool contains(String pTag) ⇒ _map.containsKey(pTag);

  // Retorna la instància enregistrada associada amb el tag o nul.
  LdTagMixin? get(String pTag) ⇒ _map[pTag];


  // 📝 CREACIÓ DE TAGS ————————————
  String newViewTag(dynamic pBase) {
    assert((pBase == null || pBase! is String || pBase! is T),
"LdTagMixin.newViewTag(pBase): pBase no és nul ni cap tipus acceptat!");
    String tag;

    if (pBase == null) {
      tag = "LdView[$_newView]";
    } else if (pBase is String) {
      tag = "$pBase[$_newView]";
    } else {
      tag = "${(pBase as LdTagIntf).baseTag}[$_newView]";
```

```
    }
    return tag;
  }

  String newWidgetTag(dynamic pBase) {
    assert((pBase! is String) && (pBase! is T));
    String tag;

    if (pBase == null) {
      tag = "LdWidget[$_newWidget]";
    } else if (pBase is String) {
      tag = "$pBase[$_newWidget]";
    } else {
      tag = "${(pBase as LdTagIntf).baseTag}[$_newView]";
    }
    return tag;
  }

  String newStateTag(dynamic pBase) {
    assert((pBase! is String) && (pBase! is T));
    String tag;

    if (pBase == null) {
      tag = "LdState[$_newState]";
    } else if (pBase is String) {
      tag = "$pBase[$_newState]";
    } else {
      tag = "${(pBase as LdTagIntf).baseTag}[$_newView]";
    }
    return tag;
  }

  String newCtrlTag(dynamic pBase) {
    assert((pBase! is String) && (pBase! is T));
    String tag;

    if (pBase == null) {
      tag = "LdCtrl[$_newCtrl]";
    } else if (pBase is String) {
      tag = "$pBase[$_newCtrl]";
    } else {
      tag = "${(pBase as LdTagIntf).baseTag}[$_newView]";
    }
    return tag;
  }
}
```

La classe *LdBinding* és responsable de crear *tags* únics per a les instàncies de *vistes*, *widgets*, *controladors* i *estats*, a partir de la base que es rep o el nom de la classe base de cada element: *LdView*, *LdWidget*, *LdCtrl* i *LdState*.

Així mateix, la classe serveix de repositori per a l'accés a totes aquestes instàncies des de qualsevol punt de l'aplicació.

Totes les instàncies que es registren han de ser desregistrades quan els seus recursos s'alliberen.

Abstracta: *LdCtrl*

```
abstract class LdCtrl<T extends LdStreamEnvelope, LdState>
with LdTagMixin
implements LdTagIntf {
  // 🧩 MEMBRES ————————————————————
  bool _isStateSet = false;
  late LdState _state;

  // 🛠️ CONSTRUCTOR/DISPOSE ——————————
  LdCtrl({ String? pTag }) {
    LdBinding bind = LdBinding.single;
    tag = bind.newCtrlTag(pTag?? baseTag);
    bind.add(tag, this);
  }

  @mustCallSuper // Arrel
  void dispose() {
    LdBinding.single.remove(tag);
  }

  // 🕹️ GETTERS/SETTERS ———————————————
  LdState get state ⇒ _state;
  set state(LdState pState) {
    assert(!_isStateSet);
    _isStateSet = true;
    _state = pState;
  }

  // ⛅ FUNCIONS ABSTRACTES ——————————
  Widget build(BuildContext pBCtx);
}
```

La classe abstracta '*LdCtrl*' és la base de totes les classes controladores de l'aplicació. Els controladors són responsables de renderitzar les vistes i els widgets que els componen així com d'actualitzar-los depenent dels events que rebin (implementant la funció abstracte '*Widget build(BuildContext pBCtx);*'.

Mixin: *LdStreamMixin*

```
mixin LdStreamMixin<E extends LdStreamEnvelope> {
  // 🧩 MEMBRES ————————————————————
  late final StreamController<E>? _streamCtrl;

  // 🛠️ LIFE CYCLE ————————————————————
  /// Inicialitza el controladors d'streams.
  void initStream() {
    _streamCtrl = StreamController<E>.broadcast();
```

```
  }

  /// Allibera els recursos associats al controlador d'streams.
  void disposeStream() {
    if (_streamCtrl ≠ null && !_streamCtrl.isClosed) {
       _streamCtrl.close();
    }
  }

  /// Retorna l'stream associat al controlador.
  Stream<E>? get stream ⇒ _streamCtrl?.stream;

  /// Subscriu un oidor a l'stream associat al controlador.
  StreamSubscription<E>? subscribe({
    required void Function(E) pLstn,
    Function? pOnError,
    void Function()? pOnDone,
    bool? cancelOnError })
    ⇒ (stream ≠ null)
      ? stream!.listen(
          pLstn,
          onError: pOnError,
          onDone: pOnDone,
          cancelOnError: cancelOnError
        )
      : null;

  /// Desubscriu un oidor de l'stream associat al controlador.
  void unsubscribe(StreamSubscription<E>? pLstn) {
    if (pLstn ≠ null) {
      pLstn.cancel();
    }
  }

  // ☁️ ESTATS ————————————————————
  /// Emet que una estructura de dades.
  void emitData<T extends LdModel>({ required String pSrcTag, String? pTgtTag, T?
pData }) {
    if (_streamCtrl == null || _streamCtrl.isClosed) return;
      _streamCtrl.add(LdModelStreamEntity<T>(pSrcTag: pSrcTag, pTgtTag: pTgtTag,
pData: pData) as E);
  }

  /// Emet que s'està preparant la càrrega de dades
  void emitPreparing({ required String pSrcTag, String? pTgtTag, bool pIsVirgin =
true}) {
    if (_streamCtrl == null || _streamCtrl.isClosed) return;
    _streamCtrl.add(LdPreparingViewStreamEvent(pSrcTag: pSrcTag, pTgtTag: pTgtTag,
pIsVirgin: pIsVirgin) as E);
  }

  /// Emet que s'està carregant les dades
  void emitLoading({ required String pSrcTag, String? pTgtTag }) {
    if (_streamCtrl == null || _streamCtrl.isClosed) return;
```

```
    _streamCtrl.add(LdLoadingStreamEntity(pSrcTag: pSrcTag, pTgtTag: pTgtTag) as
E);
  }

  /// Emet que les dades s'han carregat correctament
  void emitLoaded<D>({ required String pSrcTag, String? pTgtTag, required D data,
bool pIsVirgin = true }) {
    if (_streamCtrl == null || _streamCtrl.isClosed) return;
     _streamCtrl.add(LdLoadedStreamEntity<D>(pSrcTag: pSrcTag, pTgtTag: pTgtTag,
pData: data, pFirstTime: pIsVirgin) as E);
  }

  /// Emet que s'ha produït un error durant la càrrega
   void emitError({ required String pSrcTag, String? pTgtTag, required String
error, Exception? exception }) {
    Debug.error(error, exception);

    if (_streamCtrl == null || _streamCtrl.isClosed) return;
    _streamCtrl.add(LdErrorStreamEntity(
      pSrcTag: pSrcTag,
      pTgtTag: pTgtTag,
      pError: error,
      pException: exception
    ) as E);
  }

  /// Emet que s'està tornant a carregar les dades
  void emitReloading({ required String pSrcTag, String? pTgtTag }) {
    if (_streamCtrl == null || _streamCtrl.isClosed) return;
     _streamCtrl.add(LdReLoadingStreamEntity(pSrcTag:  pSrcTag, pTgtTag: pTgtTag)
as E);
  }

  /// Emet un canvi de tema
   void emitThemeUpdate({ required String pSrcTag, String? pTgtTag, required
ThemeData pTData }) {
    if (_streamCtrl == null || _streamCtrl.isClosed) return;
    _streamCtrl.add(LdThemeStreamEntity(
      pSrcTag: pSrcTag,
      pTgtTag: pTgtTag,
      pData:   pTData
    ) as E);
  }
}
```

El mixin '*LdStreamMixin*' és l'encarregat de crear i gestionar els streams de comunicació entre els estats i els consumidors d'events. Aquest mixin s'aplica tant a estats de vistes com a estats de widgets.

Abstracta: *LdState*

```
abstract class LdState<T extends LdStreamEnvelope, LdCtrl>
with LdTagMixin, LdStreamMixin<T>
implements LdTagIntf {
```

```
  // 🧩 MEMBRES ──────────────────────────
  bool _isVirgin = true;
  bool _isCtrlSet = false;
  late LdCtrl _ctrl;

  // 🛠️ CONSTRUCTOR/DISPOSE ──────────
  LdState({String? pTag, bool pCreateStream = true }) {
    LdBinding bind = LdBinding.single;
    tag = bind.newStateTag(pTag ?? baseTag);
    bind.add(tag, this);
    initStream();
  }

  void dispose() {
    disposeStream();
    LdBinding.single.remove(tag);
  }

  // 🕹️ GETTERS/SETTERS ──────────────
  LdCtrl get ctrl => _ctrl;
  set ctrl(LdCtrl pCtrl) {
    assert(!_isCtrlSet);
    _isCtrlSet = true;
    _ctrl = pCtrl;
  }

  // 🔄 CICLE DE CÀRREGA ─────────────
  /// Funció a implementar en cada classe filla per a la càrrega de les dades.
  Future<T?> dataProcess({ String? pSrcTag, String? pTgtTag });

  /// Executa la càrrega completa de dades
  /// Segueix el flux: Preparant → Carregant → Carregat/Error
  Future<T?> loadData({ String? pSrcTag, String? pTgtTag }) async {
    T? result;
    emitPreparing(pSrcTag: pSrcTag?? tag, pTgtTag: pTgtTag, pIsVirgin: _isVirgin);

      // Donem un petit temps perquè la UI mostri l'estat "preparant" abans de
canviar a "carregant"
    SchedulerBinding.instance.addPostFrameCallback((_) async {
      (_isVirgin)
        ? emitLoading(pSrcTag: pSrcTag?? tag, pTgtTag: pTgtTag)
        : emitReloading(pSrcTag: pSrcTag?? tag, pTgtTag: pTgtTag);
      try {
        result = await dataProcess(pSrcTag: pSrcTag,  pTgtTag: pTgtTag);
        _isVirgin = false;
        emitLoaded(pSrcTag: pSrcTag?? tag, pTgtTag: pTgtTag, data: result);
      } on Exception catch (exc) {
        String msg = "$baseTag.loadData(): ${exc.toString()}";
        emitError(
          pSrcTag:   pSrcTag?? tag,
          pTgtTag:   pTgtTag,
          error:     msg,
          exception: Exception(msg)
        );
```

```
      } on Error catch (err) {
        String msg = "$baseTag.loadData(): ${err.toString()}";
        emitError(
          pSrcTag:   pSrcTag?? tag,
          pTgtTag:   pTgtTag,
          error:     msg,
          exception: Exception(msg)
        );
      }
    });

    return result;
  }

  /// Executa una funció i captura errors
  /// Útil per operacions que no afecten l'estat principal però poden fallar
  Future<R?> safeExecute<R>({ String? pSrcTag, String? pTgtTag, required Future<R>
Function() pCBack }) async {
    try {
      return await pCBack();
    } on Exception catch (exc) {
        String msg = "$baseTag.safeExecute(pSrcpTag: ${pSrcTag?? tag}, pTgtTag:
$pTgtTag}, EXC): ${exc.toString()}";
      emitError(
        pSrcTag:   pSrcTag?? tag,
        pTgtTag:   pTgtTag,
        error:     msg,
        exception: Exception(msg)
      );
    } on Error catch (err) {
          String msg = "$baseTag.safeExecute(pTag: ${pSrcTag?? tag}, pTgtTag:
$pTgtTag}, ERR): ${err.toString()}";
      emitError(
        pSrcTag:   pSrcTag?? tag,
        pTgtTag:   pTgtTag,
        error:     msg,
        exception: Exception(msg)
      );
    }
    return null;
  }
}
```

La classe abstracta '*LdState*' és la base de totes les classes d'estat de l'aplicació (tant per a vistes com per a Widgets). És l'encarregada de l'emissió d'events a través del propi stream a mesura que es van produint.

Abstracta: *LdStreamEnvelope*

```
abstract class LdStreamEnvelope<M extends LdModel> {
  // 📝 ESTÀTICS ──────────────────────
  static const String mfTimeStamp = "mfTimeStamp";
  static const String mfSrcTag    = "mfSrcTag";
  static const String mfTgtTag    = "mfTgtTag";
```

```
  static const String mfTgtTags   = "mfTgtTags";
  static const String mfFirstTime = "mfFirstTime";

  // 🧩 MEMBRES ──────────────────────
  late final DateTime _timestamp;
  final String       _srcTag;
  final List<String> _tgtTags = [];
  final M?           _model;

  // 🕹️ GETTERS/SETTERS ──────────────
  DateTime     get timestamp ⇒ _timestamp;
  String       get srcTag    ⇒ _srcTag;
  List<String> get tgtTags   ⇒ _tgtTags;
  M?           get model     ⇒ _model;

  // 🛠️ CONSTRUCTORS ─────────────────
  LdStreamEnvelope({
    required String pSrcTag,
    List<String>? pTgtTags,
    DateTime? pTimeStamp,
    M? pModel,
  }) :
    _srcTag = pSrcTag,
    _timestamp = pTimeStamp?? DateTime.now(),
    _model     = pModel {
    _tgtTags.addAll(pTgtTags?? []);
  }

  LdStreamEnvelope.fromMap({ required LdMap<dynamic> pMap })
  : _srcTag    = pMap[mfSrcTag],
    _timestamp = ToolsDT.parse(pMap[mfTimeStamp]),
    _model     = pMap[LdModel.mfModel] {
    _tgtTags.addAll(pMap[mfTgtTags]);
  }

  // 🐑 FUNCIONS ABSTRACTES ──────────
  @mustCallSuper
  LdMap toMap()
    ⇒ LdMap({
      mfSrcTag:        _srcTag,
      mfTgtTags:       _tgtTags,
      mfTimeStamp:     ToolsDT.format(_timestamp),
      LdModel.mfModel: _model,
    });
}
```

La classe abstracta '*LdStreamEnvelope*' és l'embolcall genèric que pot enviar-se a través d'un stream de l'aplicació. A partir d'ella es descriuen tots els tipus de missatge que circulen entre emissors i oïdors d'events.

Disposa obligatòriament del tag identificador de la font de l'event així com de la llista (opcional) de tags a qui va dirigit l'event.

Abstracta: *LdModel*

```
abstract class LdModel
implements LdTagIntf {
  // 📝 ESTÀTICS ————————————————————
  static const String mfModel = "mfData";

  // 🧩 MEMBRES ————————————————————
  final String id;

  // 🛠️ CONSTRUCTOR ————————————————
  LdModel({ required this.id });

  LdModel.fromMap({ required LdMap pMap })
  : id = pMap['id'];

  // 🕹️ GETTERS/SETTERS ————————————
  @override String get baseTag;

  // ☁️ FUNCIONS ABSTRACTES ————————
  void dispose();
  LdMap toMap();
}
```

La classe abstracta '*LdModel*' és la generalització de qualsevol entitat de dades de l'aplicació. D'aquesta forma totes les estructures de dades tenen una forma comuna i poden enviar-se a través d'streams.

Classes d'Arquitectura de Vistes

L'arquitectura de les vistes aprofita les classes del *nucli* per tal de simplificar el codi per a cada vista en particular.

Abstracta: *LdViewState*

```
abstract class LdViewState<
  T  extends LdStreamEnvelope,
  VC extends LdViewCtrl<T, VC, VS>,
  VS extends LdViewState<T, VC, VS>
>
extends LdState<T, VC> {
  // 🧩 MEMBRES ──────────────────────
  String  _title;
  String? _subTitle;

  // 🕹️ GETTERS/SETTERS ──────────────
  String  get title    ⇒ _title;
  String? get subTitle ⇒ _subTitle;
  void setTitles(String pTitle, String pSubtitle) {
    _title = pTitle;
    _subTitle = pSubtitle;
    emitData(pSrcTag: tag, pTgtTag: ctrl.tag);
  }

  // 🛠️ CONSTRUCTOR/DISPOSE ──────────
  LdViewState({ required String pTitle, String? pSubtitle, super.pTag })
  : _title = pTitle,
    _subTitle = pSubtitle;

  @override
  void dispose() {
    super.dispose();
  }
}
```

La classe abstracta '*LdViewState*' és la generalització de qualsevol estat de vista de l'aplicació. Donat que totes les vistes disposen d'un títol i un subtítol opcional aquests valors els gestiona *LdViewState* simplificant l'estat de les vistes que hi derivin.

Abstracta: *LdViewCtrl*

```
abstract class LdViewCtrl<
  T extends LdStreamEnvelope,
  VC extends LdViewCtrl<T, VC, VS>,
  VS extends LdViewState<T, VC, VS>
>
extends LdCtrl<T, VS>  {
  // 🛠️ CONSTRUCTOR/DISPOSE ──────────
  LdViewCtrl({ super.pTag });

  @override
  void dispose() {
    super.dispose();
  }

  // 🌥️ 'LdCtrl' ─────────────────────
  @override
  Widget build(BuildContext pBCtx);
}
```

La classe abstracta '*LdViewCtrl*' és la generalització de qualsevol renderitzat de vista de l'aplicació.

Abstracta: *LdView*

```
abstract class LdView
extends    StatelessWidget
with       LdTagMixin
implements LdTagIntf {
  // 🧩 MEMBRES ──────────────────────
  final LdViewState _state;
  final LdViewCtrl  _ctrl;

  // 🕹️ GETTERS/SETTERS ──────────────
  LdViewState get state ⇒ _state;
  LdViewCtrl  get ctrl  ⇒ _ctrl;

  // 🛠️ CONSTRUCTOR/DISPOSE ──────────
  LdView({ super.key, String? pTag, required LdViewState pState, required
LdViewCtrl pCtrl })
  : _state = pState, _ctrl = pCtrl {
    LdBinding bind = LdBinding.single;
    tag = bind.newViewTag(pTag?? baseTag);
    bind.add(tag, this);
  }

  void dispose() {
    LdBinding.single.remove(tag);
    _state.dispose();
    _ctrl.dispose();
  }

  // 'LdTagIntf' ─────────────────────
```

```
  @override
  String get baseTag ⇒ "LdView";

  // 'StatelessState' ————————————————
  @override
  Widget build(BuildContext pBCtx) {
    return _ctrl.build(pBCtx);
  }
}
```

La classe abstracta '*LdView*' és la generalització de qualsevol vista de l'aplicació. *LdView* abstreu la gestió final dels tags i del seu registre, agrupant tant l'estat com el controlador de la vista. Un cop la vista rep la petició de construcció aquesta delega la renderització a la instància de control *LdViewCtrl*.

Classes d'Arquitectura de Widgets

L'arquitectura dels widgets o components aprofita les classes del *nucli* per tal de simplificar el codi per a cada widget en particular.

Abstracta: *LdWidgetState*

```
abstract class LdWidgetState<
  T  extends LdStreamEnvelope,
  WC extends LdWidgetCtrl<T, WC, WS>,
  WS extends LdWidgetState<T, WC, WS>
>
extends LdState<T, WC> {
  // 🛠️ CONSTRUCTOR/DISPOSE ──────────
  LdWidgetState({ required super.pTag });

  @override
  void dispose() {
    super.dispose();
  }
}
```

La classe abstracta '*LdWidgetState*' és la generalització de qualsevol estat de widget de l'aplicació.

Abstracta: *LdWidgetCtrl*

```
abstract class LdWidgetCtrl<
  T  extends LdStreamEnvelope,
  WC extends LdWidgetCtrl<T, WC, WS>,
  WS extends LdWidgetState<T, WC, WS>
>
extends LdCtrl<T, WS> {
  // 🧩 MEMBRES ──────────────────────
  final LdView _view;

  // 🛠️ CONSTRUCTOR/DISPOSE ──────────
  LdWidgetCtrl({
    required super.pTag,
    required LdView pView,
  }): _view = pView;

  @override
  void dispose() {
    super.dispose();
  }

  // 🕹️ GETTERS/SETTERS ──────────────
  LdView get view ⇒ _view;
}
```

La classe abstracta '*LdWidgetCtrl*' és la generalització de qualsevol renderitzat de widget de l'aplicació.

Abstracta: *LdWidget*

```
abstract class LdWidget
extends    StatelessWidget
with       LdTagMixin
implements LdTagIntf {
  // 🧩 MEMBRES ————————————————————
  StreamSubscription<LdStreamEnvelope>? _lstn;

  final LdWidgetState _state;
  final LdWidgetCtrl  _ctrl;
  final LdView        _view;

  // 🛠️ CONSTRUCTOR/DISPOSE ——————————
  LdWidget({
    super.key,
    String? pTag,
    required LdView pView,
    required LdWidgetState pState,
    required LdWidgetCtrl pCtrl })
  : _state = pState,
    _ctrl = pCtrl,
    _view = pView {
    LdBinding bind = LdBinding.single;
    tag = bind.newViewTag(pTag?? baseTag);
    bind.add(tag, this);
    LdBinding.single.add(tag, this);
    _lstn = _view.state.subscribe(
      pLstn: listen,
      pOnError: onError,
      pOnDone: onDone,
      cancelOnError: true,
    );

  }

  void dispose() {
    _view.state.unsubscribe(_lstn);
    LdBinding.single.remove(tag);
    _state.dispose();
    _ctrl.dispose();
  }

  // 🪆 GETTERS/SETTERS ———————————————
  LdWidgetState get state ⇒ _state;
  LdWidgetCtrl  get ctrl  ⇒ _ctrl;
  LdView        get view  ⇒ _view;

  // 🌥️ 'LdWidget' ————————————————————
  @override
  Widget build(BuildContext pBCtx) {
    return _ctrl.build(pBCtx);
  }
```

```
  // 🌥️ 'LdTagIntf' ——————————————————
  @override
  String get baseTag ⇒ "LdWidget";

  // 🌥️ FUNCIONS ESTÀTIQUES ——————————
  void listen(LdStreamEnvelope<LdModel> pEnv);
  void onError();
  void onDone();
}
```

La classe abstracta '*LdWidget*' és la generalització de qualsevol widget de l'aplicació. *LdWidget* abstreu la gestió final dels tags i del seu registre, agrupant tant l'estat com el controlador del Widget. Un cop el widget rep la petició de construcció aquesta delega la renderització a la instància de control *LdWidgetCtrl*.