
# App Architektur

*Joachim Böhmer*
*Gabriel Terwesten*

lotum


Joachim Böhmer
kaptnkoala
Gabriel Terwesten
blaugold
lotum

{

## Agenda

1. Überblick
2. Datenhaltung
3. State Management
4. Asset Verwaltung
5. Logging & Tracking

}

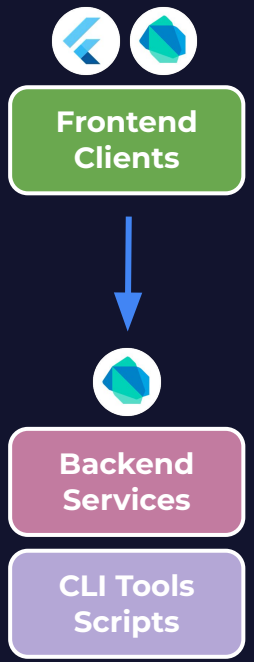
lotum

# { Überblick }

lotum

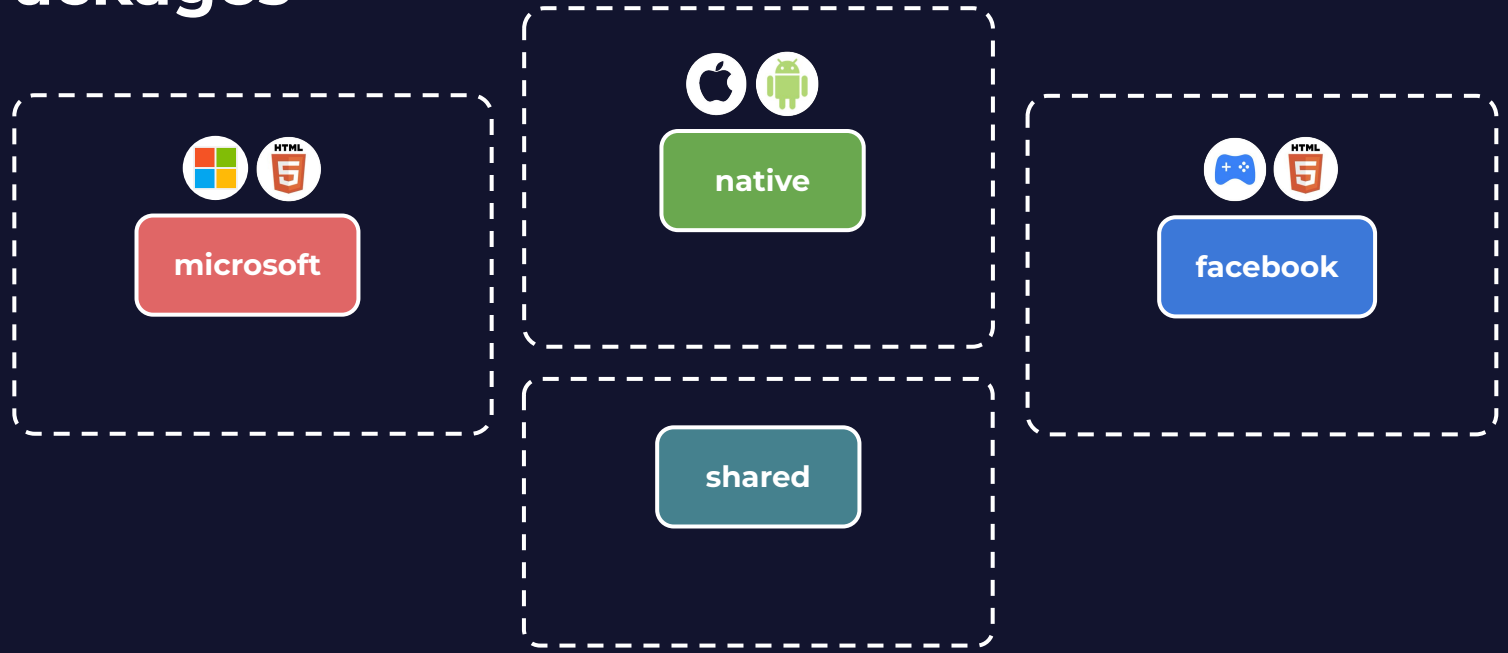
# App Architektur

- **Green Field** Projekt durch Rewrite auf Flutter
- Starker Fokus auf klare **App Architektur**
- Support für verschiedene **App Ausprägungen**
- Dart im **gesamten Stack**




lotum

# Packages

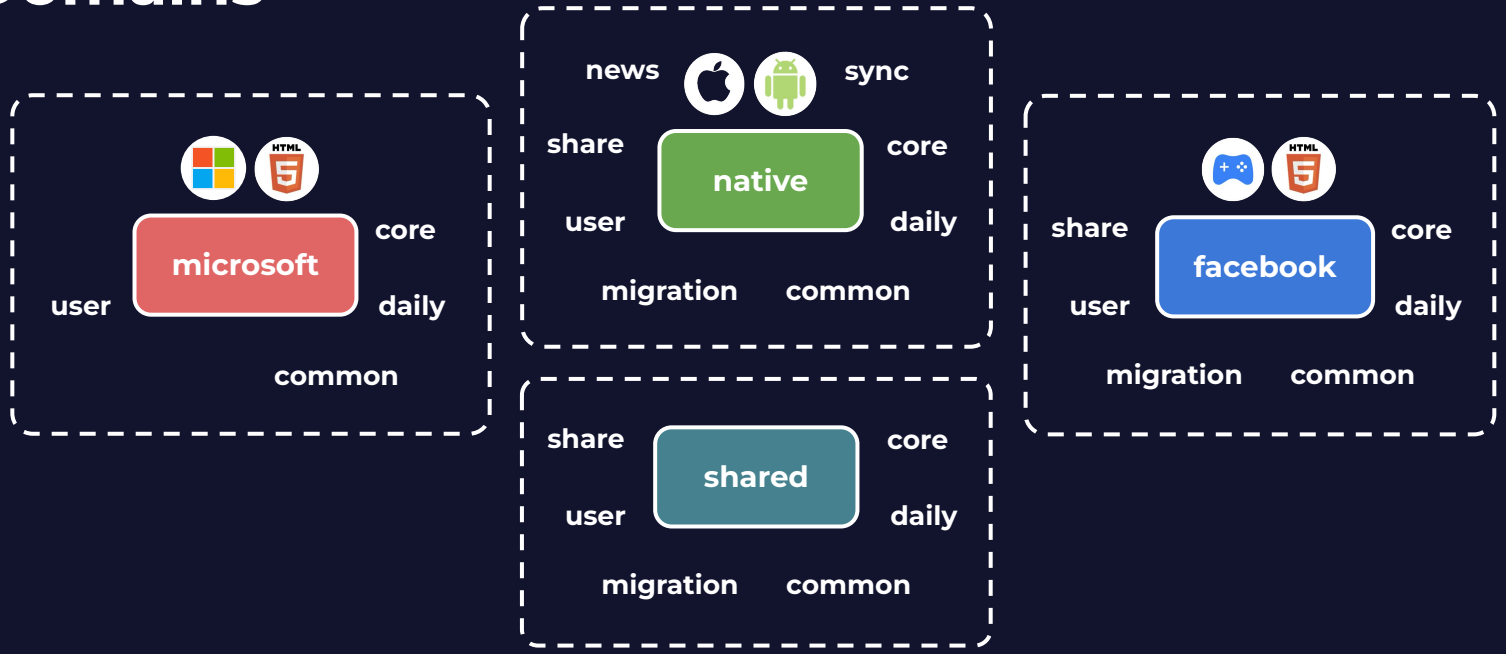



lotum

# Domains


lotum

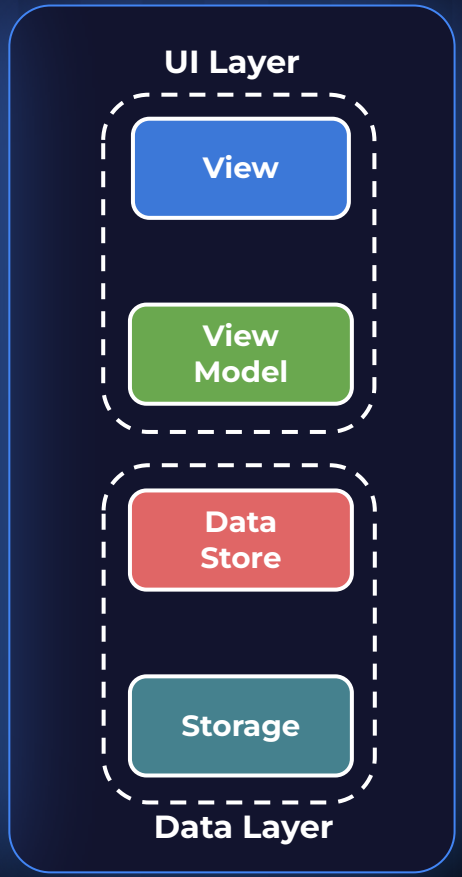
# Domains




lotum

# Abstraction Layers

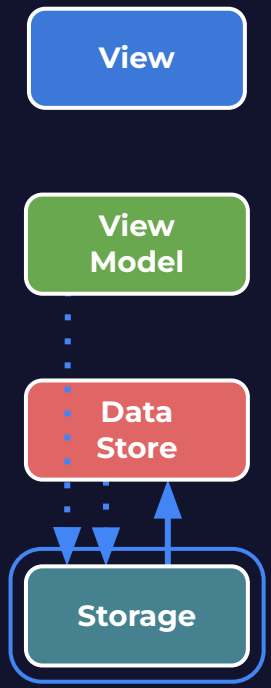
- Klare Trennung der App Layer
- Unterscheidung zwischen **Data und UI Layer**
- Abbildung in jeder **App Domain**




lotum

# Storage

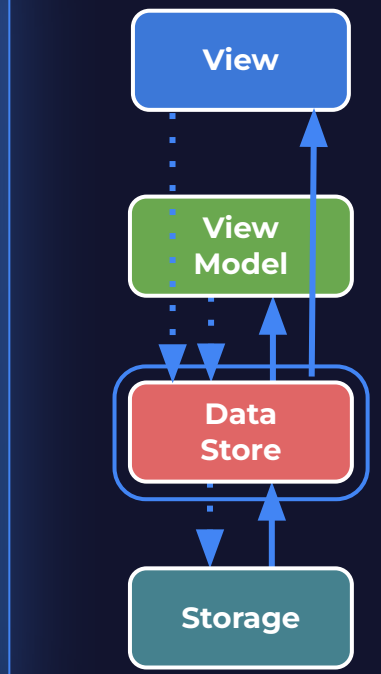
- Laden und Speichern der Daten
- **Vereinheitlichte API** über Storage Lösungen
- Zugriff erfolgt aus **Store und View Model** Ebene
- Daten explizit angefragt oder **reaktiv gepusht**




lotum

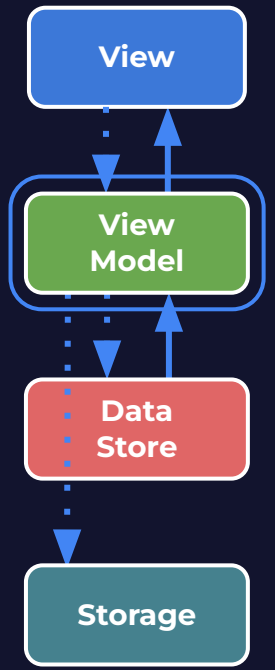
# Data Store

- Vorhalten aller relevanter App Daten
- Daten werden zu App Start initialisiert
- Daten aus Storage geholt oder subscribed
- Zugriff auf Daten von Stores oder View Models



# View Model

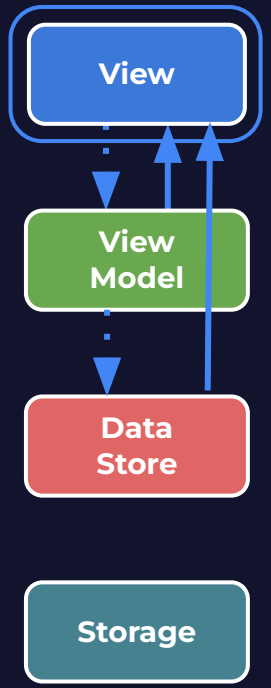
- An **Lebenszyklus** einer View gebunden
- Hält alle von der View benötigen Daten vor
- Greift auf Daten aus **Store und Storage** Ebene zu
- **Derived States** für Darstellung in View




lotum

# View

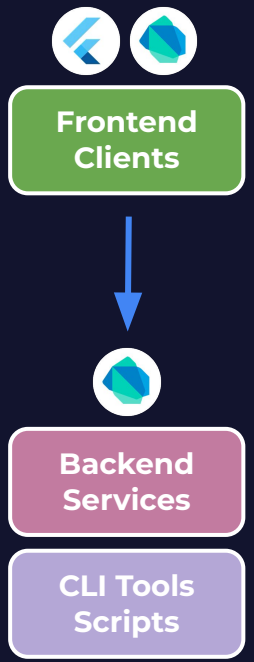
- Pages und komplexere Dialoge
- Lösen Aktionen an View Models aus
- Observen States aus View Models oder Stores
- Direkte Überführung von States in die Darstellung



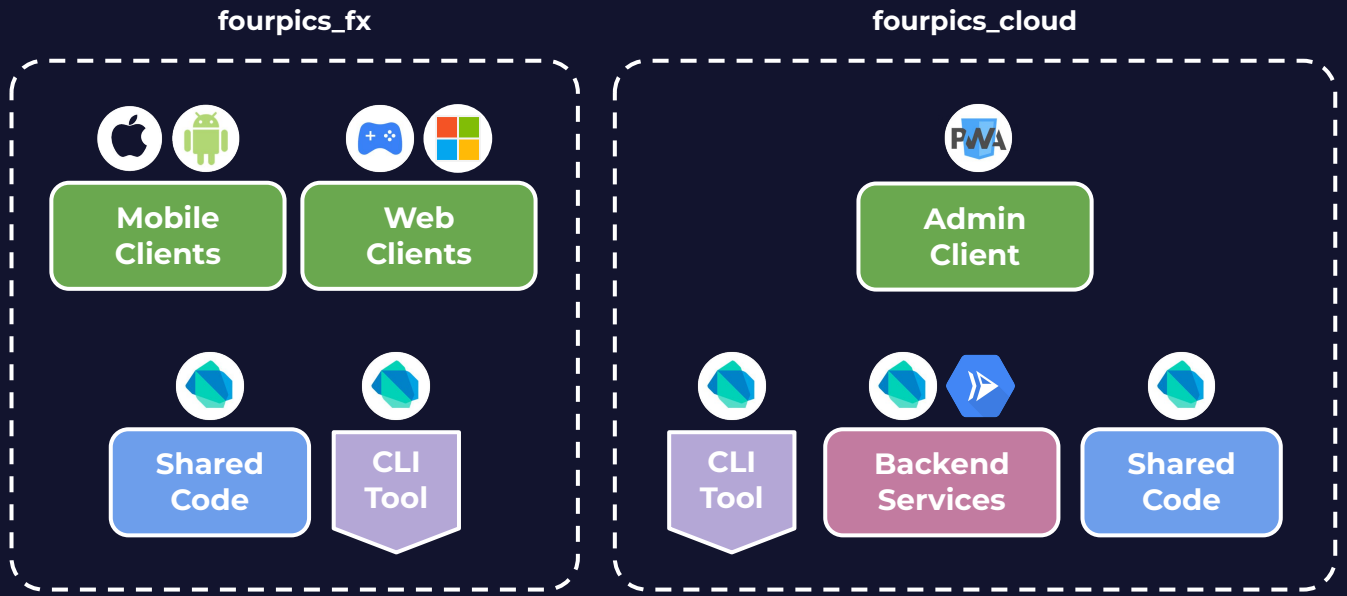

lotum

# Full Stack Dart

- Weniger **Mental Load** für Entwickler
- **Teilen von Code** zwischen Backend und Frontend
- **Performance** von Dart im Backend sehr gut
- **Package Support** überschaubar



# Mono Repositories



# { Datenhaltung }

lotum

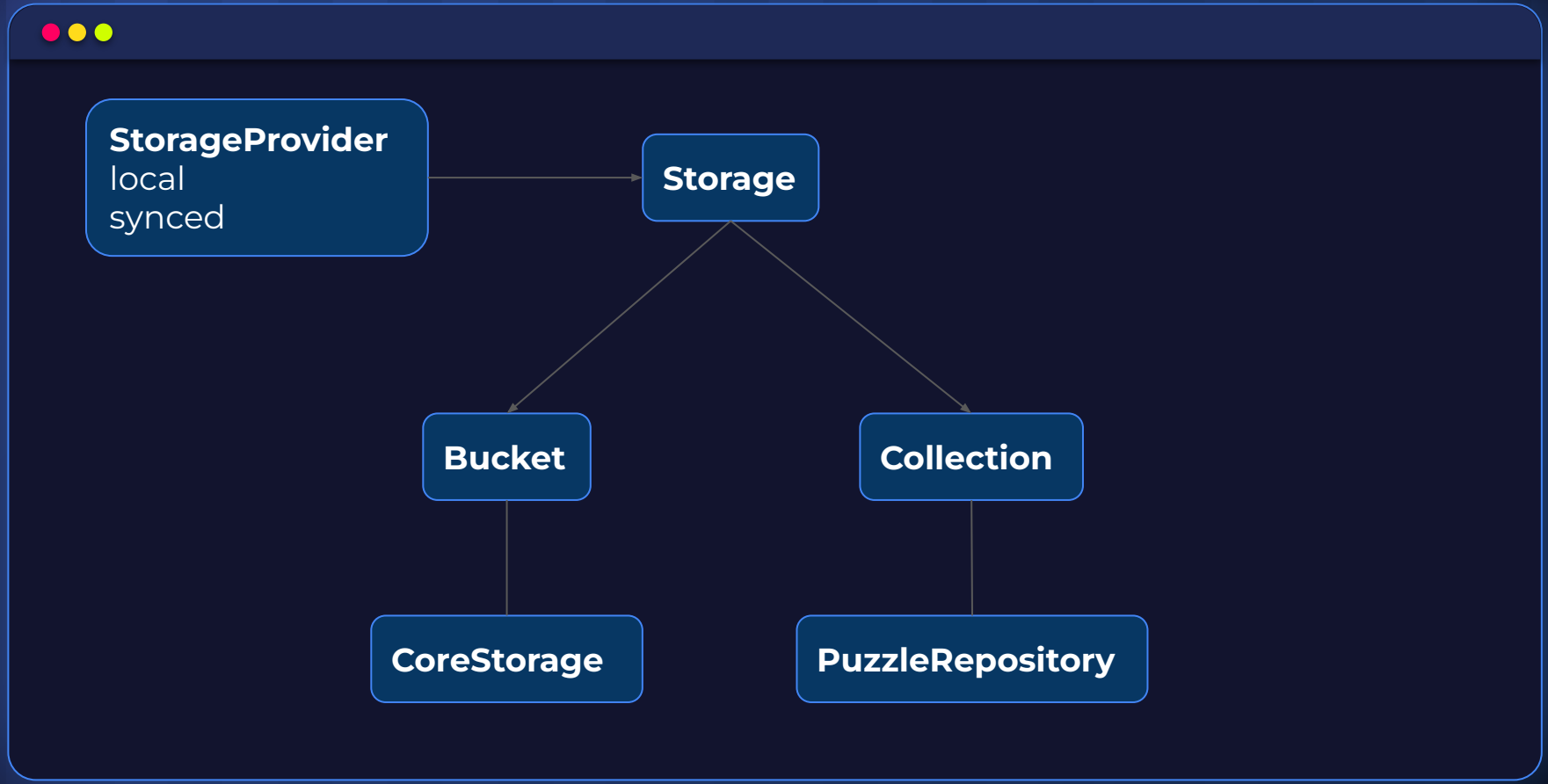
# Storage API

**Flexibilität**

**Kapselung**

**Features**

lotum


StorageProvider
local
synced
Storage
Bucket
Collection
CoreStorage
PuzzleRepository

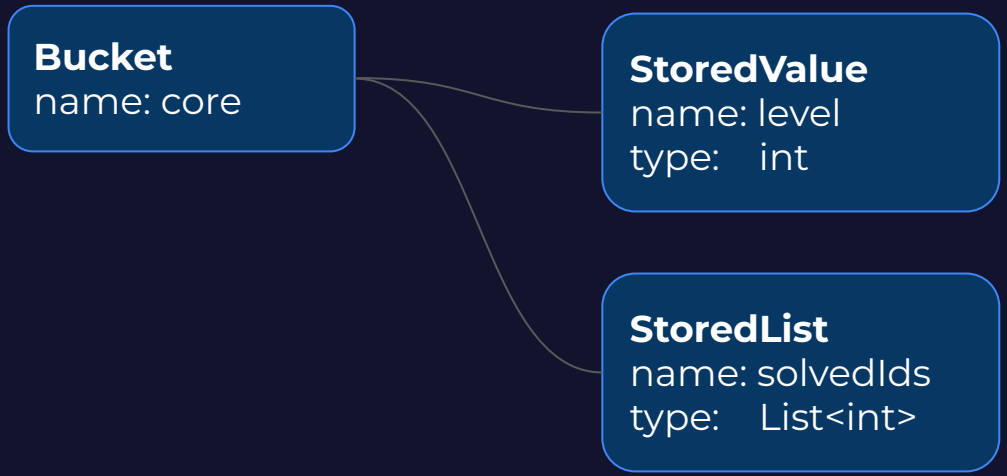
lotum

# Mapper API


```
abstract class Mapper<T, D>
abstract class ValueMapper<V> implements Mapper<V, Object>
abstract class KeyMapper<K> implements Mapper<K, String>

{Value,Key}Mapper.register<T>()
{Value,Key}Mapper.get<T>()
```

lotum

# Bucket




lotum

```dart
class CoreStorage  {
 static const _maxSolvedIds = 200;
 static const _namespace = 'core';

 factory CoreStorage(StorageProvider storage) {
   ValueMapper.register(const ListMapper(IntMapper()));

   return NativeCoreStorage._(
     storage.local.bucket(_namespace, options: const StorageOptions(lazy: true)),
     storage.synced.bucket(_namespace),
   );
 }

 CoreStorage._(this._localBucket, this._syncedBucket);

 final Bucket _localBucket;
 final Bucket _syncedBucket;

 StoredValue<int> get level => _syncedBucket.value('level', defaultValue: 1);
 StoredList<int> get solvedIds => _localBucket.list('solved_ids', capping: _maxSolvedIds);
}
```

# Collection

**Collection**
name: puzzles
type: Puzzle

**Puzzle**
id: 1

**Puzzle**
id: 2

lotum

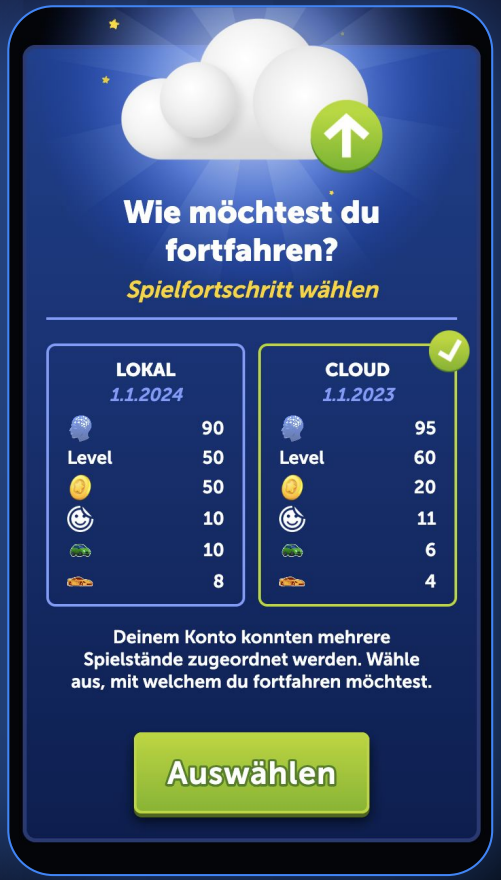
```
class PuzzleRepository {
 static const _namespace = 'puzzle';

 factory PuzzleRepository(StorageProvider storage) {
   ValueMapper.register(const PuzzleMapper());

   return PuzzleRepository._(storage.local.collection(_namespace));
 }

 PuzzleRepository._(this._puzzles);

 final Collection<int, Puzzle> _puzzles;

 Future<CorePuzzle> get(int id) => _puzzles.get(id);
 Future<List<Puzzle>> getAll() => _puzzles.values.toList();
 Future<void> save(Puzzle puzzle) => _puzzles.set(puzzle.id, puzzle);
}
```

# SyncedStorage API

**StorageConflict**
local
remote

**StorageSnapshot**
date

**StorageSyncHandler**
onResolve




lotum

# Change Notifications

- StoredValue.changes
- Collection.changesOf

# Storage Implementierung


lotum

```dart
abstract class StorageAdapter {
 const StorageAdapter(this.name);

 final String name;

 Future<int> get count;
 Stream<void> get changes;
 Stream<String> get keys;

 Future<Object?> get(String key);
 Future<Map<String, Object>>getAll(Iterable<String> keys);
 Stream<Object?> changesOf(String key);
 Future<void> set(String key, Object? value);
 Future<void> setAll(Map<String, Object?> values);
 Future<void> delete(String key);
 Future<void> deleteAll(Iterable<String> keys);
 Future<void> clear();
}
```

lotum

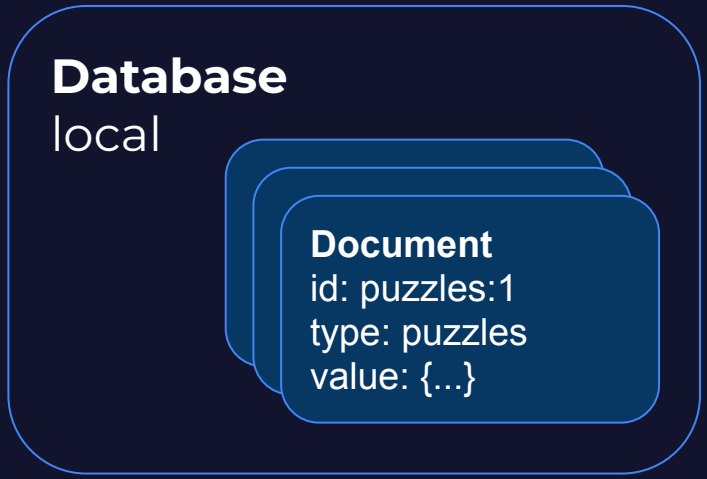
FileStorage

SqliteStorage

HiveStorage

lotum

# CouchBatchStorage




lotum

# CouchSyncedStorage

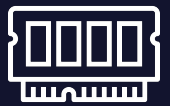
**Database**
synced

**Document**
id: game:{userId}
type: game
date: 07.10.2024
core: {
  level: 123
},
dailyProgress: {
  2024_10: {...}
}

lotum

CachingStorage

LazyPersistingStorage



FakeStorage

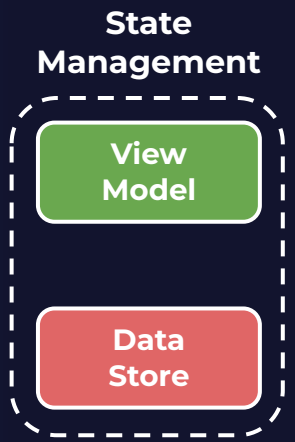

WebStorage

FacebookStorage


{ State
Management }
lotum

# State Management

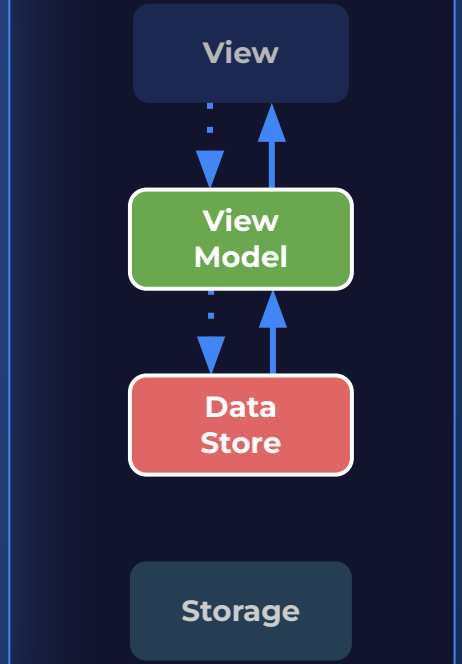
- Umfasst Data Store und View Model Ebene
- Gewährleistet **einheitlichen Zustand** der App
- Ermöglicht effizientes Update von Daten
- Verbessert die Testbarkeit der App Zustände




lotum

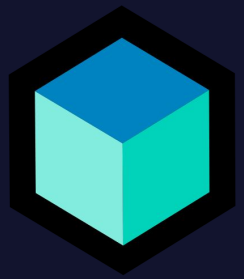
# Data Store - View Model Store

- Ausprägung des **MVVM Patterns**
- **Gleiches Reaktivitätsmodel** für beide Ebenen
- Aktionen führen Logik aus und verändern Daten
- Daten können reaktiv gepusht werden




lotum

# State Management Lösungen

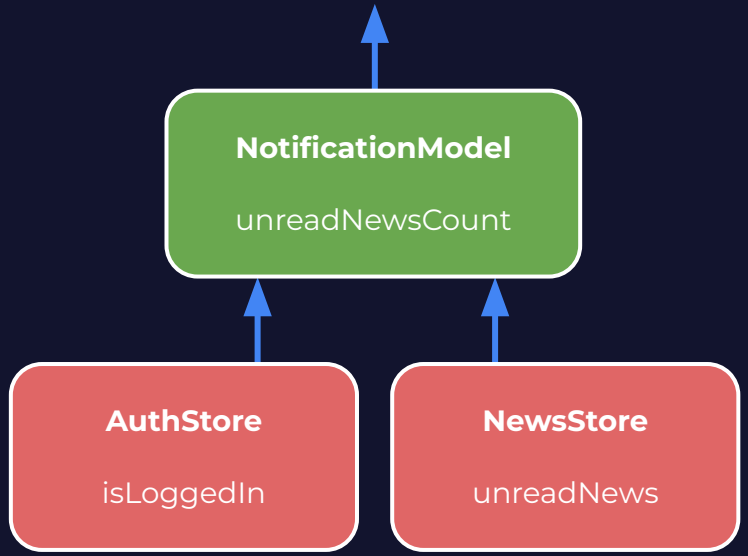


BLoC

Riverpod

MobX

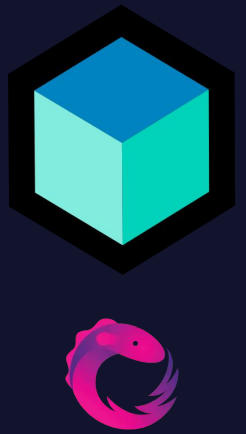
# Beispiel Szenario




lotum

# BLoC

- **Event-To-State** Pattern
- Nutzt **Cubits / Blocs**, **Events** und **States**
- Zustandänderungen über **Streams**
- Strikte Abbildung von State Management
- Optimal für große Anwendungen



lotum

```dart
class AuthState {
 final bool isLoggedIn;
}

class AuthCubit extends Cubit<AuthState> {
 AuthCubit() : super(AuthState(isLoggedIn: false));

 void logIn() => emit(AuthState(isLoggedIn: true));
}

class NewsState {
 final List<News> unreadNews;
}

class NewsCubit extends Cubit<NewsState> {
 NewsCubit() : super(NewsState(unreadNews: []);

 void readNews(News news) {
   emit(NewsState(unreadNews: List.from(state.unreadNews)..remove(news)));
 }
}

class News {
 final String title;
 final String text;
}
```

```dart
class NotificationState {
 final int unreadNewsCount;
}

class NotificationCubit extends Cubit<NotificationState> {
 late final StreamSubscription<NotificationState> _subscription;

 NotificationCubit({
   required AuthCubit authCubit,
   required NewsCubit newsCubit,
 }) : super(NotificationState(unreadNewsCount: 0)) {
   _subscription = Rx.combineLatest2<AuthState, NewsState, NotificationState>(
     authCubit.stream,
     newsCubit.stream,
     (authState, newsState) {
       return NotificationState(unreadNewsCount: authState.isLoggedIn ? newsState.unreadNews.length : 0);
     },
   ).listen(emit);
 }

 @override
 Future<void> close() {
   _subscription.cancel();
   return super.close();
 }
}
```

lotum

```dart
class BlocApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return BlocProvider<NotificationCubit>(
     create: (context) => NotificationCubit(
       authCubit: context.read<AuthCubit>(),
       newsCubit: context.read<NewsCubit>(),
     ),
     child: NewsButton(),
   );
 }
}

class NewsButton extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return BlocBuilder<NotificationCubit, NotificationState>(
     builder: (context, notificationState) {
       return BadgeIndicator(
         count: notificationState.unreadNewsCount,
         child: ImageButton(image: Assets.news.image.newsIcon),
       );
     },
   );
 }
}
```

# Riverpod

- **Provider-Based** Pattern
- Verschiedene Ausprägungen von Provider
- Sehr gut für asynchrone Programmierung
- Provider manuell oder automatisch per Annotation
- Ideal für kleine bis große Anwendungen

```dart
class AuthState {
 final bool isLoggedIn;
}

class AuthNotifier extends StateNotifier<AuthState> {
 AuthNotifier() : super(AuthState(isLoggedIn: false));

 void logIn() => state = AuthState(isLoggedIn: true);
}

class NewsState {
 final List<News> unreadNews;
}

class NewsNotifier extends StateNotifier<NewsState> {
 NewsNotifier() : super(NewsState(unreadNews: []));

 void readNews(News news) {
   state = NewsState(unreadNews: List<News>.from(state.unreadNews)..remove(news));
 }
}

// Provider instance is often located in the same file as the implementation
final authProvider = StateNotifierProvider<AuthNotifier, AuthState>((ref) {
 return AuthNotifier();
});
```

```dart
class NotificationState {
 final int unreadNewsCount;
}

final notificationProvider = Provider<NotificationState>((ref) {
 final authState = ref.watch(authProvider);
 final newsState = ref.watch(newsProvider);

 final unreadCount = authState.isLoggedIn ? newsState.unreadNews.length : 0;

 return NotificationState(unreadNewsCount: unreadCount);
});
```

```dart
class RiverpodApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return ProviderScope(
     overrides: [...],
     child: MaterialApp(
       home: NewsButton(),
     ),
   );
 }
}

class NewsButton extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return Consumer(
     builder: (context, ref, child) {
       final notificationState = ref.watch(notificationProvider);
       return BadgeIndicator(
         count: notificationState.unreadNewsCount,
         child: ImageButton(image: Assets.news.image.newsIcon),
       );
     },
   );
 }
}
```

# MobX

- **Observable-Reactive** Pattern
- Bietet **Observables, Actions** und **Derivations**
- Einfacher Code durch **Annotations**
- **Klare Modellierung** für große Anwendungen

```dart
class AuthStore = _AuthStore with _$AuthStore;

abstract class _AuthStore with Store {
 @observable
 bool isLoggedIn = false;
 @action
 void logIn() => isLoggedIn = true;
}

class NewsStore = _NewsStore with _$NewsStore;

abstract class _NewsStore with Store {
 @observable
 ObservableList<News> unreadNews = ObservableList<News>();
 @action
 void readNews(News news) => unreadNews.remove(news);
}

class News {
 final String title;
 final String text;
}
```

```dart
class NotificationModel = _NotificationModel with _$NotificationModel;

abstract class _NotificationModel with Store {
 final AuthStore authStore;
 final NewsStore newsStore;

 @computed
 int get unreadNewsCount {
   return authStore.isLoggedIn ? newsStore.unreadNews.length : 0;
 }
}

class NewsButton extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   final notificationModel = NoticationModel.of(context);
   return Observer(
     builder: (context) {
       return BadgeIndicator(
         count: notificationModel.unreadNewsCount,
         child: ImageButton(image: Assets.news.image.newsIcon),
       );
     },
   );
 }
}
```

```
mixin _$AuthStore on _AuthStore, Store {
 late final _$isLoggedInAtom = Atom(context: context);

 @override
 bool get isLoggedIn {
   _$isLoggedInAtom.reportRead();
   return super.isLoggedIn;
 }

 @override
 set isLoggedIn(bool value) {
   _$isLoggedInAtom.reportWrite(value, super.isLoggedIn, () {
     super.isLoggedIn = value;
   });
 }
}

mixin _$NotificationModel on _NotificationModel, Store {
 Computed<int> _$unreadNewsCountComputed = Computed<int>(() => super.unreadNewsCount);

 @override
 int get unreadNewsCount => _$unreadNewsCountComputed.value;
}
```

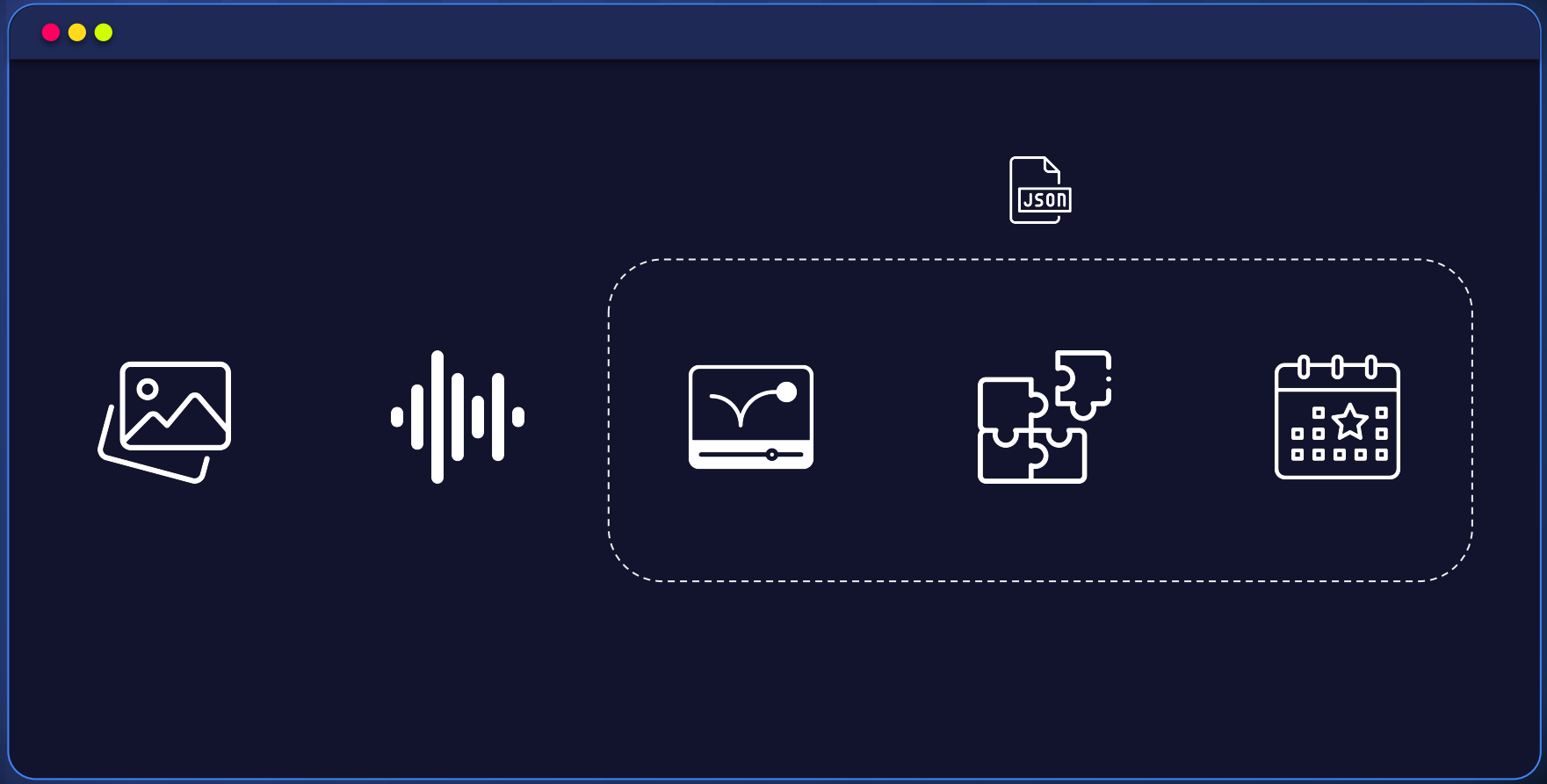
lotum

| Feature | BLoC | Riverpod | MobX |
| --- | --- | --- | --- |
| **Pattern** | Event-To-State | Provider | Observable |
| **Aufwand** | Hoch | Moderat | Niedrig |
| **Komplexität** | Moderat | Moderat | Niedrig |
| **Performance** | Gut | Sehr gut | Sehr gut |
| **Skalierbarkeit** | High | High | Moderat |

# Favorit: MobX

- Kein Boilerplate Code
- Sehr einfaches Reaktivitätsmodell
- Unterstützung von Derived States
- Kapselung von Observables in einem Store
- Sehr gut nachzuvollziehen und zu testen
- Klare Architektur für große Anwendung notwendig


{ Asset
Verwaltung }
lotum




lotum

Alle Core Puzzles
Daily Puzzles für gebündelte Monate

Bilder der ersten 250 Core Puzzles

Aktueller und kommende Monate

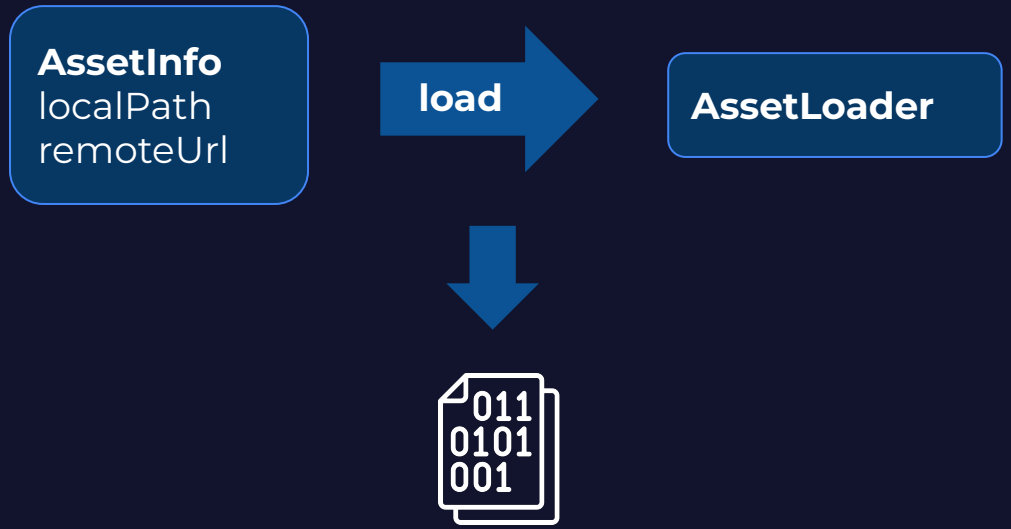
lotum

**Import Script**

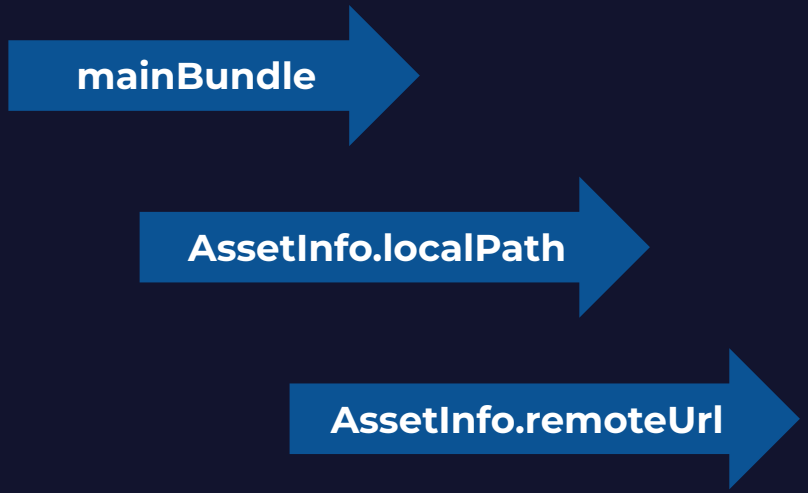
flavors
- de
  - asset
    - core
    - daily
    - puzzle
  - …
- en
- …

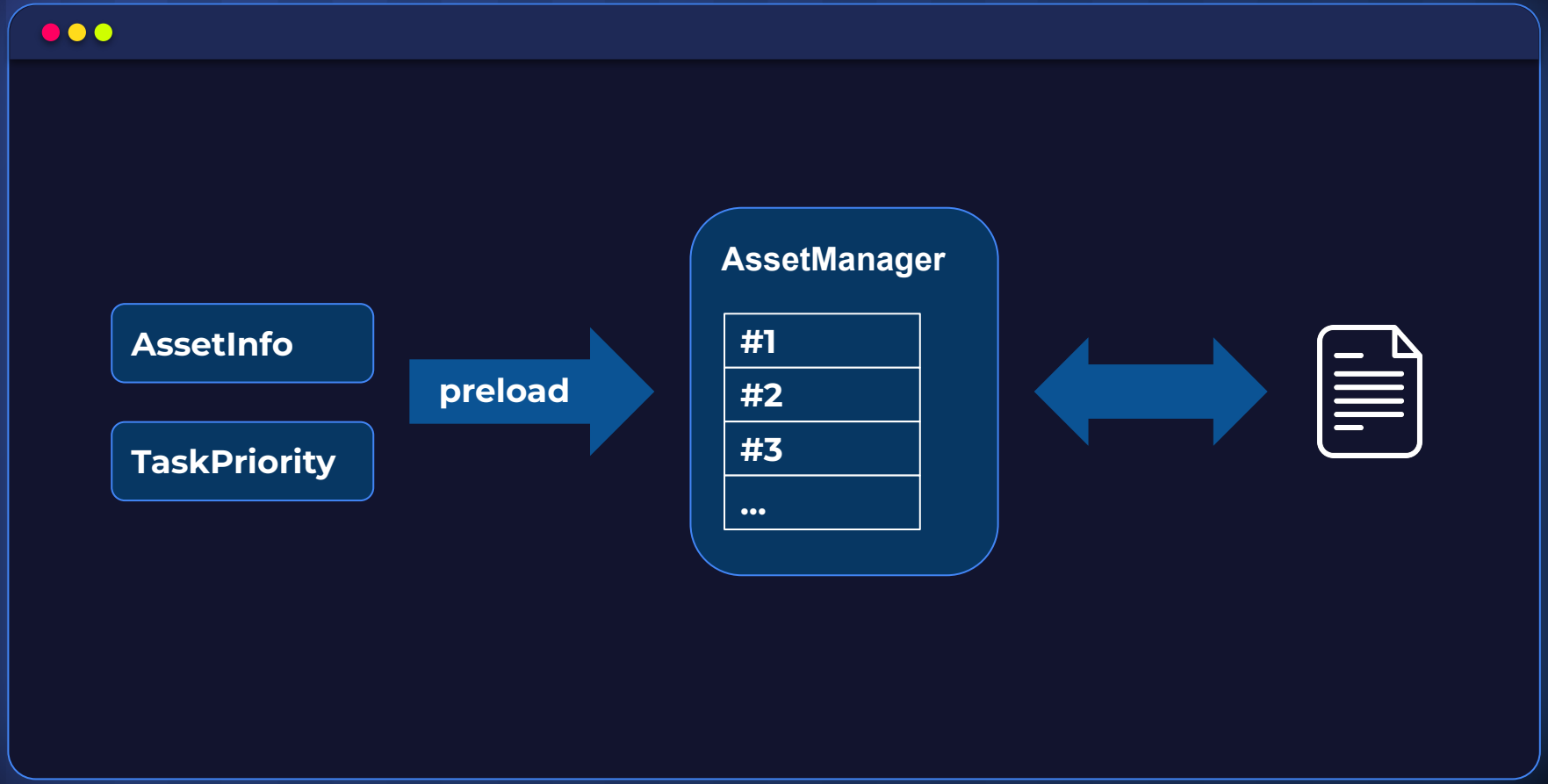
**Build Script**

asset
- core
- daily
- puzzle


AssetInfo
localPath
remoteUrl
load
AssetLoader
011
0101
001

# BundledAssetLoader

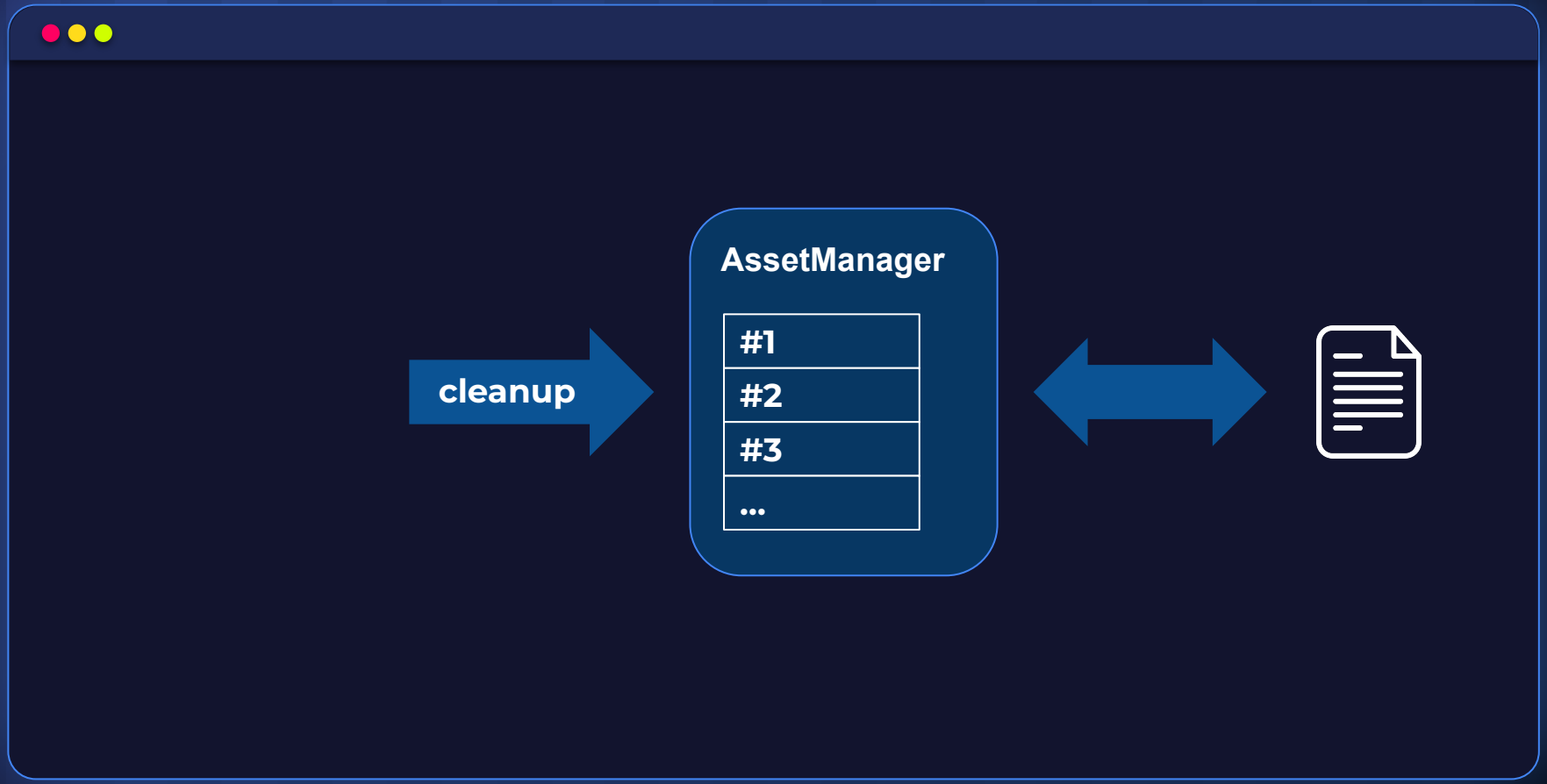




**PuzzlePreloader**
Bilder für nächsten 100 Core Puzzles

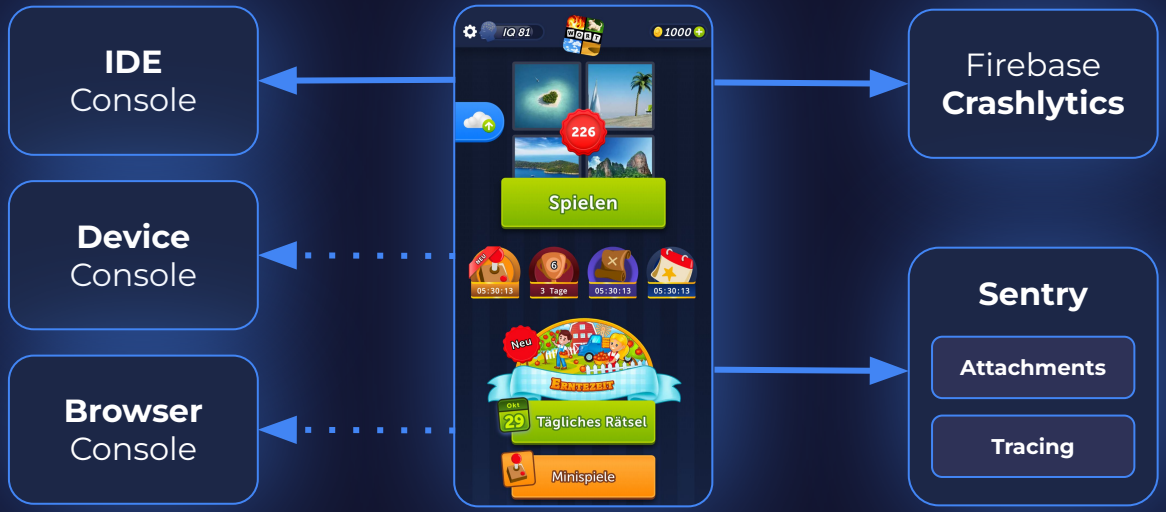
**DailyPuzzlePreloader**
Bilder für Daily Puzzles um den aktuellen Tag

**DailyAssetPreloader**
Theming Assets für alle Monate




{ Logging /
Tracking }
lotum

# Logging




lotum

```
enum LogLevel { debug, info, warn, error, fatal }

class Logger {
 final List<LogAdapter> adapters = [];
 final List<LogEntryPreprocessor> preprocessors = [];

 void debug(String message) {
   log(LogEntry(level: LogLevel.debug, message: message));
 }

 void error(Object error, StackTrace trace, {List<LogAttachment>? attachments}) {
   log(LogEntry(level: LogLevel.error, error: error, trace: trace, attachments: attachments));
 }

 void log(LogEntry entry) {
   for (final preprocessor in preprocessors) {
     entry = preprocessor(entry, this);
   }
   for (final handler in adapters) {
     handler.log(entry);
   }
 }
}
```

```dart
abstract class LogAdapter {
 void log(LogEntry entry);
}

abstract class ScopedLogAdapter extends LogAdapter {
 void setInfos(UserInfo user, DeviceInfo device);
}

class SentryLogAdapter implements ScopedLogAdapter, FlutterLogAdapter {
 @override
 void log(LogEntry entry) {
   if (entry.level case LogLevel.error || LogLevel.fatal) {
     Sentry.captureException(
       entry.error,
       stackTrace: entry.trace,
       withScope: (scope) {
         for (final attachment in entry.attachments) {
           scope.addAttachment(attachment.toSentryAttachment());
         }
       },
     );
   } else {
     Sentry.addBreadcrumb(Breadcrumb(type: 'debug', message: entry.toString()));
   }
 }
}
```

lotum

```dart
typedef LogEntryPreprocessor = LogEntry Function(LogEntry entry, Logger logger);

LogEntry parallelWaitErrorPreprocessor(LogEntry entry, Logger logger) {
 final error = entry.error;
 if (error is ParallelWaitError) {
   final errors = error.plainErrors;
   if (errors.length == 1) {
     entry = entry.copyWith(error: errors.first.error, trace: errors.first.stackTrace);
   } else {
     for (final error in errors) {
       logger.error(error.error, error.stackTrace);
     }
   }
 }
 return entry;
}

extension on ParallelWaitError<void, Object?> {
 List<AsyncError> get plainErrors {
   return switch (errors) {
     final List<AsyncError?> errors => errors,
     (final AsyncError? e0, final AsyncError? e1) => [e0, e1],
     (final AsyncError? e0, final AsyncError? e1, final AsyncError? e2) => [e0, e1, e2],
   }.whereNotNull().toList();
 }
}
```

# Tracking


Firebase **Analytics**

Google **BigQuery**

**BI ETL** Workflows

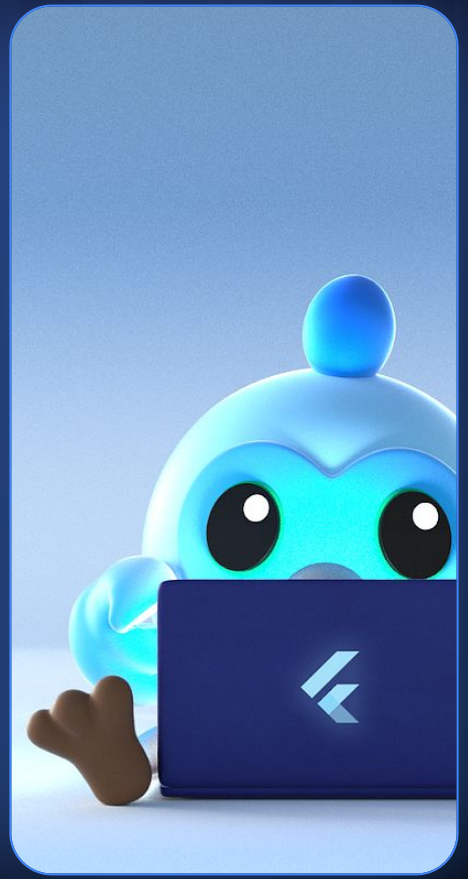
**Custom** Analytics Solutions

lotum

```
enum AnalyticsChannel { firebase, ... }

class Analytics {
 final List<AnalyticsAdapter> adapters = [];

 void setUserProperties(Map<String, Object> properties) {
   for (final adapter in adapters) {
     adapter.setUserProperties(properties);
   }
 }

 void log(String event, {Map<String, Object>? params, Iterable<AnalyticsChannel>? channels}) {
   for (final adapter in adapters.where((adapter) => adapter.matches(channels))) {
     adapter.log(event, params);
   }
 }
}

abstract class AnalyticsAdapter {
 bool matches(Iterable<AnalyticsChannel> channels);
 void setUserProperties(Map<String, Object> properties);
 void log(String event, Map<String, Object>? params);
}
```

# Fazit

- Klare Architektur erforderlich
- Flexibilität durch Daten Abstraktion
- Nachvollziehbarkeit im State Management
- Intensive Tests von View Models sinnvoll
- Dart im gesamten Stack ist super



lotum

**Danke**
**für eure**
**Aufmerksamkeit.** 💙

lotum