ГЛАВА 6

Займемся данными по-настоящему

Данные — непростая и очень обширная тема: их структура и взаимосвязи с другими данными; варианты обработки, хранения и извлечения; различные стандарты; поставщики и движки баз данных и многое другое. Возможно, данные — самый сложный аспект разработки, с которым разработчики сталкиваются и в самом начале карьеры, и позднее при изучении любого нового набора инструментов.

Дело в том, что без данных в какой-либо форме практически все приложения теряют смысл. Очень немногие приложения приносят хоть какую-то пользу без хранения, извлечения или установления взаимосвязей данных.

Будучи основой практически всего ценного, что приносят приложения, *данные* становятся причиной множества инноваций, вносимых поставщиками баз данных и платформ. Но во многих случаях сложность никуда не девается — в конце концов, это весьма глубокая и обширная тема.

И тут на сцену выходит Spring Data (https://spring.io/projects/spring-data). Заявленная цель Spring Data: «Обеспечить привычную и единообразную модель программирования на основе Spring для доступа к данным с сохранением при этом всех характеристик используемого хранилища данных». Вне зависимости от движка базы данных или платформы, цель Spring Data — сделать для разработчика доступ к данным наиболее простым и полнофункциональным.

В этой главе описывается хранение и извлечение данных на основе различных промышленных стандартов и ведущих баз данных, а также проекты и возможности Spring Data, позволяющие использовать их наиболее простым и полнофункциональным способом — посредством Spring Boot.

Описание сущностей

Практически во всех случаях, когда мы имеем дело с данными, применяется какая-либо сущность предметной области: счет-фактура ли это, автомобиль или что-то иное, — редко используют данные в виде набора несвязанных свойств. То, что мы считаем полезными данными, неизбежно представляет собой взаимосвязанные совокупности элементов, которые все вместе составляют нечто осмысленное. Автомобиль — в виде данных или реальный — полезен лишь тогда, когда обладает всеми нужными уникальными атрибутами.

Spring Data предоставляет несколько различных механизмов и вариантов доступа к данным для приложений Spring Boot на самых разных уровнях абстракции. Вне зависимости от уровня абстракции, выбранного разработчиком для конкретного сценария использования, первый шаг — описание классов предметной области, которые будут использоваться для обработки соответствующих данных.

И хотя полноценное обсуждение предметно-ориентированного проектирования (Domain-Driven Design, DDD) выходит за рамки темы книги, мы воспользуемся некоторыми его идеями в качестве основы для описания нужных классов предметной области в примерах приложений в этой и следующих главах. Полное описание DDD можно найти, например, в посвященном этой теме фундаментальном труде Эрика Эванса (Eric Evans) *Domain-Driven Design: Tackling Complexity in the Heart of Software* (https://oreil.ly/DomainDrivDes)[1].

Если объяснять в общих словах, *класс предметной области* (domain class) инкапсулирует основную сущность предметной области, обладающую значимостью независимо от других данных. Это не означает, что она не связана с другими сущностями предметной области, а говорит лишь о том, что она самодостаточна и имеет смысл как отдельная единица даже без привязки к другим сущностям.

Для создания класса предметной области в Spring с помощью Java можно создать класс с переменными-членами, соответствующими конструкторами, методами доступа и изменения, методами `equals()`/`hashCode()`/`toString()` и многими другими. Можно также воспользоваться Lombok вместе с Java или классами данных в Kotlin, чтобы создавать классы предметной области для представления, хранения и извлечения данных. Все вышеупомянутое я проделаю в этой главе, чтобы продемонстрировать, насколько просто работать с предметной областью с помощью Spring Boot и Spring Data. Чем больше вариантов, тем лучше.

---

[1] *Эванс Э.* Предметно-ориентированное проектирование (DDD): структуризация сложных программных систем. — М.: Вильямс, 2010.

Определив для примеров из этой главы класс предметной области, я выберу базу данных и уровень абстракции в соответствии с целями использования данных и предоставляемых API для базы данных. В экосистеме Spring это сводится обычно к одному из двух вариантов (с небольшими вариациями): шаблоны или репозитории.

Поддержка шаблонов

Spring Data, чтобы предоставить набор связных абстракций довольно высокого уровня, описывает интерфейсы типа `Operations` для большинства различных источников данных. Эти интерфейсы `Operations`, например `MongoOperations`, `RedisOperations` и `CassandraOperations`, задают базовый набор операций, которые можно применять напрямую для достижения максимальной гибкости или создавать на их основе абстракции более высокого уровня. Классы `Template` обеспечивают прямые реализации интерфейсов `Operations`.

Шаблоны можно считать своего рода интерфейсами поставщиков сервисов (Service Provider Interface, SPI). Они обладают очень большими возможностями и допускают непосредственное использование, но каждый раз требуют повтора множества шагов для реализации наиболее распространенных сценариев использования, встречающихся разработчикам. Для сценариев, в которых доступ к данным имеет типичные закономерности, возможно, больше подойдут репозитории. А лучше всего то, что в основе репозиториев лежат шаблоны, так что вы ничего не потеряете от перехода к абстракции более высокого уровня.

Поддержка репозиториев

В Spring Data определен интерфейс `Repository`, базовый для всех остальных типов интерфейсов репозиториев Spring Data, например `JPARepository` и `MongoRepository`, предоставляющих ориентированные на JPA и Mongo возможности соответственно, и различные многофункциональные интерфейсы наподобие `CrudRepository`, `ReactiveCrudRepository` и `PagingAndSortingRepository`. В этих многофункциональных интерфейсах описано много полезных высокоуровневых операций, например `findAll()`, `findById()`, `count()`, `delete()`, `deleteAll()` и др.

Репозитории определены как для блокирующих, так и для неблокирующих взаимодействий. Кроме того, репозитории Spring Data поддерживают создание запросов в стиле «соглашения важнее конфигурации» и даже точных операторов

запросов. Использование репозиториев Spring Data со Spring Boot упрощает реализацию сложных взаимодействий с базами данных практически до уровня тривиальной задачи.

Я продемонстрирую все эти возможности далее в книге. В этой главе собираюсь охватить ключевые элементы нескольких вариантов баз данных путем сочетания различных деталей реализации: Lombok, Kotlin и др. Таким образом я создам широкий и прочный фундамент для последующих глав.

@Before

Как бы сильно я ни любил кофе и ни полагался на него при разработке приложений, изучать рассматриваемые в остальной части книги идеи будет удобнее на более универсальной предметной области. Я не только разработчик программного обеспечения, но и пилот, поэтому полагаю, что все более сложный и управляемый данными мир авиации в избытке предоставит интересные ситуации и захватывающие данные для исследования, когда мы будем углубляться в механизмы Spring Boot в многочисленных сценариях использования.

Чтобы работать с данными, прежде всего нужны *данные*. Я разработал небольшой реализующий REST веб-сервис `PlaneFinder` (доступен в прилагаемом к книге репозитории кода) в качестве API-шлюза для запросов о воздушных судах, находящихся в пределах досягаемости маленького устройства на моем столе, и их местоположении. Это устройство получает данные автоматического наблюдения — передачи данных (Automatic Dependent Surveillance Broadcast, ADS-B) от самолетов, находящихся в пределах определенного расстояния, и передает их онлайн-сервису PlaneFinder.net (https://planefinder.net). Также оно предоставляет API HTTP, данные от которого мой шлюз потребляет, упрощает и открывает для использования расположенными далее по конвейеру сервисами наподобие представленных в этой главе.

Далее вас ожидает больше подробностей, а пока что создадим несколько сервисов, подключаемых к базе данных.

Создание с помощью Redis сервиса на основе шаблона

Redis — база данных, используемая обычно в качестве хранилища данных в оперативной памяти для обмена информацией о состоянии между различными

экземплярами сервиса, кэширования и в качестве брокера сообщений между сервисами. Подобно многим основным базам данных, Redis способна на большее, но в этой главе мы воспользуемся ею лишь для хранения и извлечения из памяти информации о воздушных судах, получаемой нашим сервисом от вышеупомянутого сервиса `PlaneFinder`.

Инициализация проекта

Для начала обратимся к Spring Initializr. Я выбрал следующие опции:

- проект Maven;
- Java;
- текущая стабильная версия Spring Boot;
- упаковка — JAR;
- Java — 11.

И зависимости:

- Spring Reactive Web (`spring-boot-starter-webflux`);
- Spring Data Redis (Access+Driver) (`spring-boot-starter-data-redis`);
- Lombok (`lombok`).

ПРИМЕЧАНИЯ ОБ ОПЦИЯХ ПРОЕКТА

В скобках после названия меню Initializr указаны идентификаторы артефактов для приведенных ранее возможностей/библиотек. У первых двух — общий идентификатор группы — `org.springframework.boot`, а у Lombok — `org.projectlombok`.

И хотя я не выбирал умышленно какие-либо неблокирующие реактивные функциональные возможности для приложений этой главы, я включил зависимость для Spring Reactive Web вместо Spring Web, чтобы получить доступ к WebClient — предпочтительному клиенту как для блокирующих, так и для неблокирующих взаимодействий приложений, основанных на Spring Boot 2.0 и более поздних версий, с сервисами. С точки зрения разработчика, создающего простейший веб-сервис, код не меняется, какую бы из этих зависимостей я ни включил: в примерах этой главы код, аннотации и свойства совершенно одинаковы для обоих. Я укажу все различия по мере того, как в следующих главах два этих пути начнут расходиться.

Далее сгенерируем проект, сохраним его на локальной машине, разархивируем и откроем в IDE.

Разработка сервиса Redis

Начнем с предметной области.

В настоящее время API-шлюз `PlaneFinder` предоставляет одну конечную точку REST:

```
http://localhost:7634/aircraft
```

Любой локальный сервис может обратиться с запросом к этой конечной точке и получить ответ в виде JSON со списком всех воздушных судов, находящихся в пределах досягаемости приемника, в следующем формате (с типичными образцами данных):

```
[
    {
        "id": 108,
        "callsign": "AMF4263",
        "squawk": "4136",
        "reg": "N49UC",
        "flightno": "",
        "route": "LAN-DFW",
        "type": "B190",
        "category": "A1",
        "altitude": 20000,
        "heading": 235,
        "speed": 248,
        "lat": 38.865905,
        "lon": -90.429382,
        "barometer": 0,
        "vert_rate": 0,
        "selected_altitude": 0,
        "polar_distance": 12.99378,
        "polar_bearing": 345.393951,
        "is_adsb": true,
        "is_on_ground": false,
        "last_seen_time": "2020-11-11T21:44:04Z",
        "pos_update_time": "2020-11-11T21:44:03Z",
        "bds40_seen_time": null
    },
    {<другое воздушное судно в пределах досягаемости, те же поля, что и выше>},
    {<последнее воздушное судно, находящееся сейчас в пределах досягаемости,
    те же поля, что и выше>}
]
```

Определение класса предметной области

Для ввода и обработки отчетов о воздушных судах я создал класс `Aircraft` следующего вида:

```
package com.thehecklers.sburredis;

import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import com.fasterxml.jackson.annotation.JsonProperty;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.springframework.data.annotation.Id;

import java.time.Instant;

@Data
@NoArgsConstructor
@AllArgsConstructor
@JsonIgnoreProperties(ignoreUnknown = true)
public class Aircraft {
    @Id
    private Long id;
    private String callsign, squawk, reg, flightno, route, type, category;
    private int altitude, heading, speed;
    @JsonProperty("vert_rate")
    private int vertRate;
    @JsonProperty("selected_altitude")
    private int selectedAltitude;
    private double lat, lon, barometer;
    @JsonProperty("polar_distance")
    private double polarDistance;
    @JsonProperty("polar_bearing")
    private double polarBearing;
    @JsonProperty("is_adsb")
    private boolean isADSB;
    @JsonProperty("is_on_ground")
    private boolean isOnGround;
    @JsonProperty("last_seen_time")
    private Instant lastSeenTime;
    @JsonProperty("pos_update_time")
    private Instant posUpdateTime;
    @JsonProperty("bds40_seen_time")
    private Instant bds40SeenTime;

    public String getLastSeenTime() {
        return lastSeenTime.toString();
    }

    public void setLastSeenTime(String lastSeenTime) {
        if (null != lastSeenTime) {
            this.lastSeenTime = Instant.parse(lastSeenTime);
        } else {
            this.lastSeenTime = Instant.ofEpochSecond(0);
        }
```

```
    }

    public String getPosUpdateTime() {
        return posUpdateTime.toString();
    }

    public void setPosUpdateTime(String posUpdateTime) {
        if (null != posUpdateTime) {
            this.posUpdateTime = Instant.parse(posUpdateTime);
        } else {
            this.posUpdateTime = Instant.ofEpochSecond(0);
        }
    }

    public String getBds40SeenTime() {
        return bds40SeenTime.toString();
    }

    public void setBds40SeenTime(String bds40SeenTime) {
        if (null != bds40SeenTime) {
            this.bds40SeenTime = Instant.parse(bds40SeenTime);
        } else {
            this.bds40SeenTime = Instant.ofEpochSecond(0);
        }
    }
}
```

Этот класс предметной области включает несколько полезных аннотаций, упрощающих код и/или повышающих его гибкость. В число аннотаций уровня класса входят следующие:

- `@Data` — указывает Lombok создать метод-геттер, метод-сеттер, `equals()`, `hashCode()` и `toString()`, то есть так называемый класс данных;
- `@NoArgsConstructor` — позволяет создать конструктор без параметров, то есть не требующий аргументов;
- `@AllArgsConstructor` — позволяет создать конструктор с параметром для каждой переменной-члена, то есть требующий указания аргумента для каждого из них;
- `@JsonIgnoreProperties(ignoreUnknown = true)` — указывает механизмам десериализации Jackson игнорировать поля в JSON-ответах, для которых не существует соответствующей переменной экземпляра.

Аннотации уровня полей позволяют точнее контролировать поведение. В числе примеров аннотаций уровня полей — две аннотации этого класса:

- `@Id` — помечает аннотированную переменную-член класса как содержащую уникальный идентификатор записи базы данных;
- `@JsonProperty("vert_rate")` — связывает переменную-член класса с полем JSON с другим именем.

Вам может быть интересно, зачем я создал явным образом методы доступа и изменения для трех переменных экземпляра типа `Instant`, если аннотация `@Data` обеспечивает создание методов-геттеров и методов-сеттеров для всех переменных экземпляра. В случае этих трех переменных необходимо выполнить синтаксический разбор JSON-значения и преобразовать его из `String` в составной тип данных с помощью вызова метода `Instant::parse`. Если значения вообще нет (`null`), нужно следовать другой логике, чтобы не передать случайно `null` методу `parse()` и чтобы присвоить осмысленное подстановочное значение соответствующей переменной-члену с помощью метода-сеттера. Кроме того, сериализацию значения типа `Instant` лучше всего выполнять путем преобразования в `String` — отсюда и необходимость в явных методах-геттерах.

Теперь, описав класс предметной области, мы можем создать и настроить механизм доступа к базе данных Redis.

Добавляем поддержку шаблонов

Spring Boot предоставляет базовую функциональность `RedisTemplate` посредством автоконфигурации, и чтобы выполнять операции над значениями `String` с помощью Redis, нужно совсем немного работы и кода. Работа же со сложными объектами предметной области требует чуть более обширной конфигурации.

Класс `RedisTemplate` расширяет класс `RedisAccessor` и реализует интерфейс `RedisOperations`. Для нашего приложения особенно интересен `RedisOperations`, поскольку в нем описана необходимая для взаимодействия с Redis функциональность.

Разработчики должны стараться писать код для интерфейсов, а не для реализаций, поскольку это позволяет использовать наиболее подходящую реализацию для решения поставленной задачи без изменения кода или API или излишних либо ненужных нарушений принципа DRY (Don't Repeat Yourself — «Не повторяйся»). Если интерфейс реализован полностью, то любая конкретная реализация работает не хуже остальных.

В следующем листинге я создаю компонент типа `RedisOperations`, возвращающий `RedisTemplate` в качестве конкретной реализации компонента. Чтобы

настроить его должным образом для работы с входящим[1] `Aircraft`, я сделал следующее.

1. Создал `Serializer` для преобразования между объектами и записями JSON. Поскольку для маршалинга/демаршалинга (сериализации/десериализации) значений JSON применяется Jackson, уже включенный в веб-приложения Spring Boot, я создал `Jackson2JsonRedisSerializer` для объектов типа `Aircraft`.
2. Создал объект `RedisTemplate`, принимающий ключи типа `String` и значения типа `Aircraft`, для работы с получаемым на входе объектом `Aircraft` с идентификаторами типа `String`. А также подключил компонент `RedisConnectionFactory`, автоматически связываемый с единственным параметром метода создания этого компонента, `RedisConnectionFactory factory`, к объекту `template`, что позволяет ему создавать и получать соединение с базой данных Redis.
3. Передал сериализатор `Jackson2JsonRedisSerializer<Aircraft>` объекту `template` в качестве сериализатора по умолчанию. У `RedisTemplate` есть несколько сериализаторов, в качестве которых может использоваться сериализатор по умолчанию, если им не был присвоен какой-то конкретный, что очень удобно.
4. Создал еще один сериализатор для ключей, чтобы шаблон не пытался использовать сериализатор по умолчанию, ожидающий объекты типа `Aircraft`, для преобразования в значения ключей типа `String` и обратно. Для этого отлично подходит `StringRedisSerializer`.
5. Наконец, возвратил созданный и настроенный объект `RedisTemplate` в качестве компонента, который должен применяться, когда внутри приложения требуется какая-нибудь реализация компонента `RedisOperations`:

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.data.redis.connection.RedisConnectionFactory;
import org.springframework.data.redis.core.RedisOperations;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.data.redis.serializer.Jackson2JsonRedisSerializer;
import org.springframework.data.redis.serializer.StringRedisSerializer;

@SpringBootApplication
public class SburRedisApplication {
    @Bean
    public RedisOperations<String, Aircraft>
```

[1] В оригинале игра слов — inbound может означать и «входящий», и «прилетающий». — *Примеч. пер.*

```
    redisOperations(RedisConnectionFactory factory) {
        Jackson2JsonRedisSerializer<Aircraft> serializer =
                new Jackson2JsonRedisSerializer<>(Aircraft.class);

        RedisTemplate<String, Aircraft> template = new RedisTemplate<>();
        template.setConnectionFactory(factory);
        template.setDefaultSerializer(serializer);
        template.setKeySerializer(new StringRedisSerializer());

        return template;
    }

    public static void main(String[] args) {
        SpringApplication.run(SburRedisApplication.class, args);
    }
}
```

Собираем все вместе

Мы подготовили фундамент для доступа к базе данных Redis с помощью шаблона, пришло время пожинать плоды. Как показано в следующем листинге, я создал класс Spring Boot, снабженный аннотацией `@Component`, для опроса конечной точки `PlaneFinder` и обработки записей `Aircraft`, полученных благодаря поддержке шаблонов Redis.

Для инициализации компонента `PlaneFinderPoller` и подготовки его к работе я создал объект `WebClient`, указывающий на целевую конечную точку, предоставляемую внешним сервисом `PlaneFinder`, и присвоил его переменной экземпляра. В настоящее время сервис `PlaneFinder` работает на моей локальной машине и прослушивает на порте 7634.

Компоненту `PlaneFinderPoller` для работы необходим доступ к двум другим компонентам: `RedisConnectionFactory` (предоставляется средствами автоконфигурации Spring Boot, поскольку одна из зависимостей приложения — Redis) и реализации интерфейса `RedisOperations` — созданному ранее `RedisTemplate`. Оба присваиваются описанным должным образом переменным-членам посредством внедрения зависимости через конструктор (с автоматическим связыванием):

```
import org.springframework.data.redis.connection.RedisConnectionFactory;
import org.springframework.data.redis.core.RedisOperations;
import org.springframework.scheduling.annotation.EnableScheduling;
import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.client.WebClient;

@EnableScheduling
@Component
class PlaneFinderPoller {
```

```
    private WebClient client =
            WebClient.create("http://localhost:7634/aircraft");

    private final RedisConnectionFactory connectionFactory;
    private final RedisOperations<String, Aircraft> redisOperations;

    PlaneFinderPoller(RedisConnectionFactory connectionFactory,
                      RedisOperations<String, Aircraft> redisOperations) {
        this.connectionFactory = connectionFactory;
        this.redisOperations = redisOperations;
    }
}
```

Далее я создал метод, в котором и выполняется основная работа. Для реализации опроса по заданному расписанию воспользовался аннотацией `@EnableScheduling`, размещенной ранее на уровне класса, и снабдил созданный мною метод `pollPlanes()` аннотацией `@Scheduled`, передав в нее параметр `fixedDelay=1000`, указывающий, что опрос выполняется каждые 1000 мс (один раз в секунду). Остальная часть этого метода состоит лишь из трех декларативных операторов: для очистки ранее сохраненных объектов `Aircraft`, извлечения и сохранения текущих местоположений самолетов и вывода отчета о последней захваченной информации.

Что касается первой из упомянутых задач: я воспользовался автоматически связываемым объектом `Connectionfactory` для получения соединения с базой данных, через которое выполнил команду сервера по очистке всех имеющихся ключей — `flushDb()`.

Второй оператор вызывает `PlaneFinder` с помощью объекта `WebClient` и извлекает набор воздушных судов в пределах досягаемости вместе с информацией об их местоположении в данный момент. Тело ответа преобразуется в `Flux` объектов `Aircraft`, фильтруется для исключения объектов `Aircraft` без регистрационных номеров, преобразуется в `Stream` объектов `Aircraft` и сохраняется в базу данных Redis. Сохранение каждого допустимого объекта `Aircraft` производится присваиванием паре «ключ — значение» регистрационного номера `Aircraft` и самого объекта `Aircraft` соответственно с помощью предназначенных для манипулирования значениями данных операций Redis.

Flux — реактивный тип данных, он описан в следующих главах, а пока что можете считать его просто набором объектов, поставляемых без блокирования.

Последний оператор в `pollPlanes()` снова использует несколько операций над значениями, определенных в Redis, для извлечения всех ключей (с помощью

подстановочного знака *), извлечения соответствующего каждому из ключей значения Aircraft и последующего вывода на экран. Вот законченная версия метода pollPlanes():

```
@Scheduled(fixedRate = 1000)
private void pollPlanes() {
    connectionFactory.getConnection().serverCommands().flushDb();

    client.get()
            .retrieve()
            .bodyToFlux(Aircraft.class)
            .filter(plane -> !plane.getReg().isEmpty())
            .toStream()
            .forEach(ac -> redisOperations.opsForValue().set(ac.getReg(), ac));

    redisOperations.opsForValue()
            .getOperations()
            .keys("*")
            .forEach(ac ->
                System.out.println(redisOperations.opsForValue().get(ac)));
}
```

Окончательная (на данный момент) версия класса PlaneFinderPoller приведена в следующем листинге:

```
import org.springframework.data.redis.connection.RedisConnectionFactory;
import org.springframework.data.redis.core.RedisOperations;
import org.springframework.scheduling.annotation.EnableScheduling;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.client.WebClient;

@EnableScheduling
@Component
class PlaneFinderPoller {
    private WebClient client =
            WebClient.create("http://localhost:7634/aircraft");

    private final RedisConnectionFactory connectionFactory;
    private final RedisOperations<String, Aircraft> redisOperations;

    PlaneFinderPoller(RedisConnectionFactory connectionFactory,
                      RedisOperations<String, Aircraft> redisOperations) {
        this.connectionFactory = connectionFactory;
        this.redisOperations = redisOperations;
    }

    @Scheduled(fixedRate = 1000)
    private void pollPlanes() {
```

```
        connectionFactory.getConnection().serverCommands().flushDb();

        client.get()
                .retrieve()
                .bodyToFlux(Aircraft.class)
                .filter(plane -> !plane.getReg().isEmpty())
                .toStream()
                .forEach(ac ->
                    redisOperations.opsForValue().set(ac.getReg(), ac));

        redisOperations.opsForValue()
                .getOperations()
                .keys("*")
                .forEach(ac ->
                    System.out.println(redisOperations.opsForValue().get(ac)));
    }
}
```

Механизмы опроса полностью реализованы, и мы можем запустить приложение и посмотреть, что получилось.

Результаты

С предварительно запущенным на моей машине сервисом `PlaneFinder` я запустил приложение `sbur-redis` для получения, сохранения и извлечения информации из Redis, а также отображения на экране результатов каждого из опросов сервиса `PlaneFinder`. Далее приведен пример полученных результатов, отредактированный для краткости и немного отформатированный для удобства чтения:

```
Aircraft(id=1, callsign=EDV5015, squawk=3656, reg=N324PQ, flightno=DL5015,
route=ATL-OMA-ATL, type=CRJ9, category=A3, altitude=35000, heading=168,
speed=485, vertRate=-64, selectedAltitude=0, lat=38.061808, lon=-90.280629,
barometer=0.0, polarDistance=53.679699, polarBearing=184.333345, isADSB=true,
isOnGround=false, lastSeenTime=2020-11-27T18:34:14Z,
posUpdateTime=2020-11-27T18:34:11Z, bds40SeenTime=1970-01-01T00:00:00Z)

Aircraft(id=4, callsign=AAL500, squawk=2666, reg=N839AW, flightno=AA500,
route=PHX-IND, type=A319, category=A3, altitude=36975, heading=82, speed=477,
vertRate=0, selectedAltitude=36992, lat=38.746399, lon=-90.277644,
barometer=1012.8, polarDistance=13.281347, polarBearing=200.308663, isADSB=true,
isOnGround=false, lastSeenTime=2020-11-27T18:34:50Z,
posUpdateTime=2020-11-27T18:34:50Z, bds40SeenTime=2020-11-27T18:34:50Z)

Aircraft(id=15, callsign=null, squawk=4166, reg=N404AN, flightno=AA685,
route=PHX-DCA, type=A21N, category=A3, altitude=39000, heading=86, speed=495,
vertRate=0, selectedAltitude=39008, lat=39.701611, lon=-90.479309,
barometer=1013.6, polarDistance=47.113195, polarBearing=341.51817, isADSB=true,
isOnGround=false, lastSeenTime=2020-11-27T18:34:50Z,
posUpdateTime=2020-11-27T18:34:50Z, bds40SeenTime=2020-11-27T18:34:50Z)
```