ervices

Reactive microservices is an activator template completely devoted to microservices architecture. It lets you learn about microservices in general — different patterns, communication protocols and 'tastes' of microservices. All these concepts are demonstrated using Scala, Akka, Play and other tools from Scala ecosystem. For the sake of clarity, we skipped topics related to deployment and operations — that's a great subject for another big activator template.

- See the prerequisities you will need to fully understand the template.
- See the build info needed to run the template.
- If you wan't to know more about microservices, see: what are microservices section.
- If you're interested in the systems architecture details, see: the system blueprint of the tutorial.

1.1. Prerequisites

To feel comfortable while playing with this template, make sure you know basics of Akka HTTP which is a cornerstone of this project. We recently released an Akka HTTP activator template that may help you start. At least brief knowledge of Akka remoting, Akka persistence,Akka streams and Play Framework websockets is also highly recommended. Anyway, don't worry — all these technologies (and more!) will be discussed on the way but we won't dig into the details.

- Go to build overview

1.2. Project build overview

This activator template consists of 9 runnable subprojects — the microservices:

- auth ones:
  - auth-codecard
  - auth-fb
  - auth-password
  - identity-manager
  - session-manager
  - token-manager
- business logic ones:
  - btc-users
  - btc-ws
- miscellaneous ones:
  - metrics-collector

They uses different communication methods, different databases, and different frameworks.

Go to setup instructions.

1.3. Setup instructions

Review the configuration files

Take some time to review `application.conf` files that are located in `resource` subdirectory of each microservice. You can also look at `docker-compose.yml` file, which contains docker preconfigurated images for all the required databases.

Run migrations (You don't need to do this step if you want to use our docker container)

For `auth-codecard`, `identity-manager` and `auth-password` you need to run the SQL migration scripts which are located in postgres directory. If you want to use non-default names please tweak the \`application.conf\` files. You can also tweak and use this script in your console:

```
cd /where/this/activator/template/is/located/
psql -h localhost -U postgres -f ./postgres/init.sql &&
psql -h localhost -U postgres -f ./postgres/auth_entry.sql &&
psql -h localhost -U postgres -f ./postgres/identity.sql
```

Running

Run `docker-compose` up in project main directory to launch databases, or if you are using your own database instances, make sure you have PostgreSQL, MongoDB and Redis up and running.

Akka HTTP

You can run each service separately, but we also we provided a SBT task called `runAll`.

Play

Due to some issues with Play/sbt cooperation `metrics-collector` and `btc-ws` should be run separately. In order to run them in one sbt CLI instance use these commands:

```
; project btc-ws; run 9000
; project metrics-collector; run 5001
```

Everything else should work out of the box. Enjoy!

See what are microservices,

or the system blueprint.

2. What are microservices?

There's no formal or widespread definition of a microservice. However usually microservices are defined as an software architectural style in which system is composed of multiple services. Those services are small (or at least smaller than in typical, monolith applications), can be independently deployed and they communicate using (lightweight) protocols. Well–defined microservice should be organized around business capabilities or to put it in Domain–Driven Design wording — should encapsulate Bounded Context. Sometimes it is said that microservices are implementation of Single Responsibility Principle in architecture.

See opportunities and dillemas of using microservice architecture.

2.1. Opportunities

Applications based on a microservice architecture are basically distributed systems. This means that they can scale horizontally in a much more flexible way than monolithic systems — instead of replicating whole heavyweight process one can spawn multiple instances of services that are under load. This guarantees better hardware utilization — money savings. Another important consequence of moving from monolith to microservices is the need of designing for failure which can result in a truly reactive system. Like it or not, while designing a distributed system you have to take failure into account — otherwise you will see your system falling apart.

However, technical benefits of introducing a microservice–based architecture are far less important than the social ones. MSA enables truly cross–functional teams organized around business capabilities which are implemented by microservices. This in turn allows easy team scaling and makes team members care more about actual business behind the technology. This agility together with autonomy of teams usually result in a shorter time–to–market. In fact, most of the early adopters of microservices (Netflix, Amazon, SoundCloud, Tumblr) underline the ability to deliver faster as the main selling point of microservices. Shorter time–to–market stems not only from well–organized teams but also from other features of MSA. First of all, microservices are supposed to be easier to understand (thus maintain) than monolithic systems. Smaller size also means that microservices can be easily rewritten (or simply disposed) instead of being refactored which usually is expansive and results in sub–optimal code quality. Autonomy of teams enables polyglot approach which provides better utilization of tools and bounded context is supposed to increase code reusability. Microservices should also be fun for developers, as everything that's new and challenging.

See the dillemas and problems with microservice architecture

2.2. Dilemmas and problems

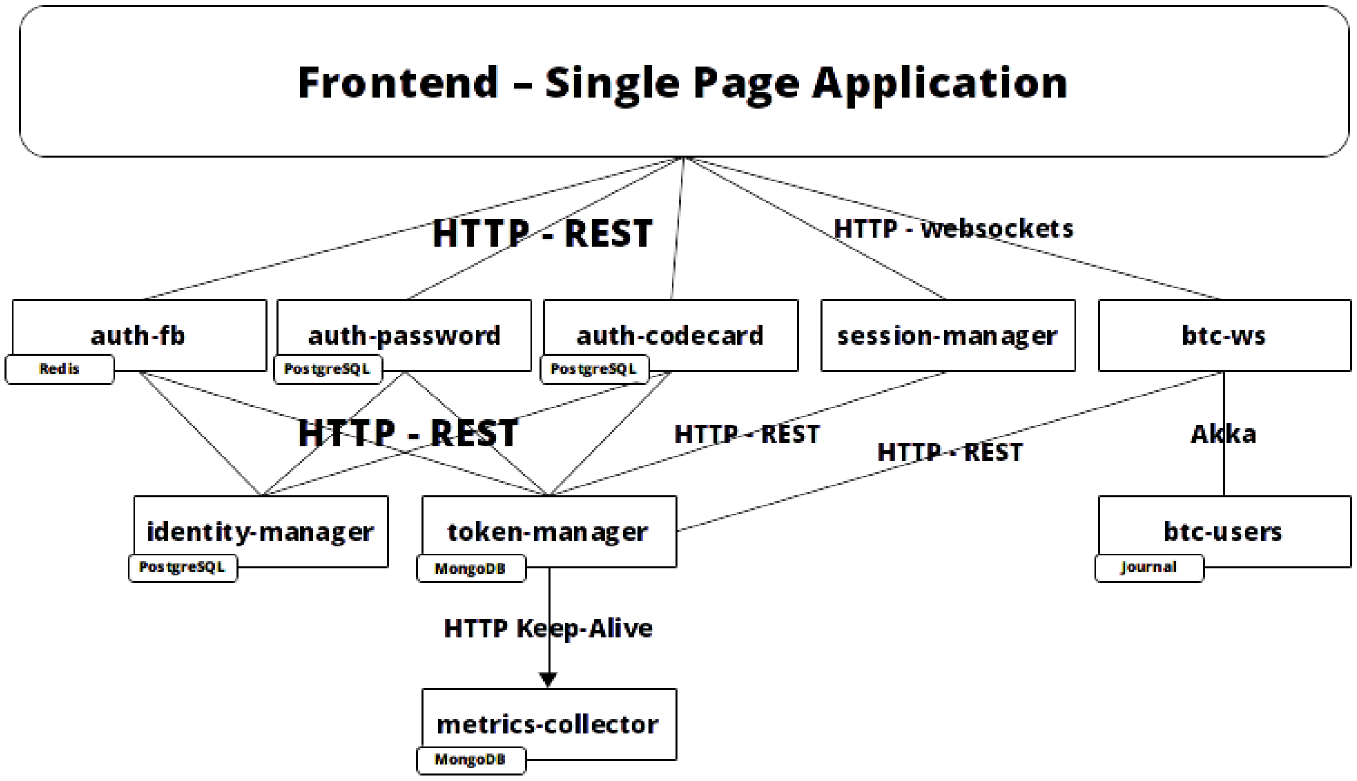
A shift from monolithic to microservices architecture is a serious step. Distributed systems are totally different than monolithic ones and have their own dilemmas and problems. First and foremost microservices are all about communication and protocols. One should be aware that microservices doesn't magically suppress complexity — they just move it from code to communication layer. Different communication protocols (synchronous and asynchronous) and transport guarantees will be the main subject of this tutorial (see chapters 3.1., 3.2., 3.3., 3.4.). Another important subject worth mentioning while discussing microservices is a polyglot persistence — how to embrace multiple different data stores and not lose consistency and performance. You can check out this approach in our activator template in chapter 3. Very common question asked while developing microservices is 'how big is a microservice?' — we'll discuss it in chapter 3. Microservice approach requires some boilerplate; one may be tempted to share code using shared libraries which introduce coupling — why and when would you like to do that? See chapter 3.1. There's also a multitude of the problems which are out of the scope of this template like: testing (why, when and how to do it?), polyglot approach (is it worth the cost?), operations, contract management (what's my API, who are my collaborators, how can I contact them?), API versioning (how to stay backward compatible?), logging & debugging and security. Feel encouraged to enrich this activator template with suitable examples and tutorials — and let us know!

See the system blueprint

3. System blueprint

To present different concepts related to microservices we built an authentication system. The idea behind it is really simple — you can sign in/sign up using arbitrarily chosen authentication methods (currently they're email–password, Facebook Oauth, codecard). Number of used authentication methods indicates user's token strength. User that presents a valid authentication token can access business applications behind the authentication system. To test if our authentication system actually works we integrated a simple application that after singing in lets users subscribe and get notifications in real–time about bitcoin market events such as rate change, volume above/below certain level etc.

System design



If you want to have a closer look at this schema go here.

Complete system consists of 9 microservices:

identity-manager — serves as a identity issuer in our system; it is being called each time user registers.

token-manager — focal point of the system; manages tokens which consist of identity id, value and expiration date; it is being called when user signs in, signs out or other service verifies identity of token supplier.

metrics-collector — gathers important metrics from critical *token-manager* service; allows to preview them using websockets.

session-manager — proxies a requests to token-manager that come from the outside (untrusted) world.

auth-fb — provides authentication via Facebook Oauth; it is being called when user signs in/signs up using this method.

auth-password — provides authentication using email/password pair; it is being called when user signs in/signs up using this method.

auth-codecard — provides authentication with card of codes; it is being called when user signs in/signs up using this method.

btc-users — manages subscriptions; fetches data from BTC market; routes events to users; communicates with *btc-ws*.

btc-ws — serves as a façade and a translation layer over *btc-users* offering end–user convenient websockets API; communicates with *token-manager* to authenticate user access.

Spend some time analyzing communication paths, protocols and data stores — it might be useful during our tour around each service of the system.

See the synchronous HTTP communication method,

the asynchronous HTTP streams communication,

the asynchronous HTTP websockets communication.

or the asynchronous Akka message communication.

3.1. Synchronous HTTP — akka–http

Synchronous communication is a data transfer method in which receiving and sending are governed by strict timing signals. It's usually takes the form of a request–response protocol — party sends data only when it's explicitly asked for it. That's how typical HTTP service works. Synchronous protocols are very popular because they're easy to understand and analyze. However they have one significant drawback — they don't scale well. First of all, synchronous protocols introduce liveness issues — when you ask for something you have to be prepared that your call may explicitly fail or even worse you may never get any response at all (that's why you need timeouts). Secondly, usually if you ask for something, you have to wait for the response to continue processing. Several such calls and you'll end up waiting most of the time instead of doing something actually useful. Having said that, world wide web we use daily is mostly synchronous — you click 'Log in' button and you wait for 'Login successful' confirmation box. Request–response — that's how vanilla HTTP works. Synchronous communication is also useful (well, almost unavoidable) while designing microservice–based system — learn why and when.

identity-manager

identity-manager is a microservice for issuing and managing existing identities in the system. Code of this service is very straightforward, it consists of:

- Slick functional mapping definition
- Extracting config values
- Database connection & configuration
- Little repository layer
- Routing

identity-manager has well–defined responsibility — creating and listing existing identities. For now it only keep identity id and creation date but as such system evolves it might be useful to keep ex. user's first and last name (feel encouraged to create this enhancement!). The service is really short (75 lines!), thus easy to understand and maintain. It doesn't have well–known layer structure (repository–service–routing) as it would only introduce burden and additional boilerplate. That's however not always the case — let's see more complex services.

auth-codecard, auth-password

auth-codecard, auth-password are services for signing in/signing up using codecard and email–password respectively. These are user facing services, which means that SPA frontend application communicates with them directly. They're synchronous by design — when user logs in or registers, she has to wait for the operation to complete before proceeding. These services are much more complex, thus structured:

- Initialization & routing logic which calls methods of services (codecard, password)
- JsonProtocols which are being used to convert objects to jsons (codecard, password)
- Config values which are used in different parts of the application (codecard, password)
- Service that handles business logic by interacting with repository (local datastore) and gateway (remote calls) (codecard, password)
- Repository that handles interactions with local datastore (codecard, password)
- Gateway that communicates with external services (identity-manager & token-manager) (codecard, password)

There are a few things to note about these services. First of all, they provide similar functionalities but each authentication method has its own characteristics (ex. codecard needs card renewal, email–password needs password changing) — that's why they're completely separated. Another important thing is that structure introduces organizational complexity but abstraction allows to hide some implementation details and makes code easier to understand and maintain. Last but not least, careful reader might have noticed that both Gateway source codes looks really similar (codecard, password). One may be tempted to write a shared library to reduce source code duplication. Shared libraries are generally discouraged when writing microservices because they introduce another layer of coupling. However, that's sometimes unavoidable — we'll see that in the next chapter.

auth-fb

auth-fb is a service for signing in/signing up with Facebook Oauth and is very similar to auth-codecard and auth-password. However it misses separate *Repository* layer. Microservices are all about agility and flexibility, so we don't have to follow the same rules all the time. auth-fb uses Redis — database with very simple interface and writing dedicated class for interacting with it would be an overkill. That's why all the database access is done inside Service. Looking at auth-fb it's easy to notice how polyglot persistence approach is helping developing microservices — each authentication method service is responsible for (and only for) its own registration data and can use the best persistence method to store it (simple key–value store in case of auth-fb). So called 'distributed truth' makes systems little harder to maintain and use (you can't query all the datastores at once, data access is non–uniform etc.) but it enables better tools utilization, scaling, decoupling and resilience (of course when done right).

token-manager

token-manager is a focal point of our authentication system. Authentication method services gets a fresh token for logged in users from it and business services verifies tokens presented by users to check identities. token-manager is built using previously presented layered architecture (routes — service — repository) but it misses gateway as it doesn't initiate communication with other services. It's the most important service in whole project so it's equipped with custom metrics reporting — there's processing time reporting for every requestand success/failure reporting for each action. You'll learn how it works under the hood in the next chapter.

session-manager

In the microservice architecture you usually have services that offer public API accessible by clients and internal API that is being used only by other services. Sometimes service offers some features that are public and others that are strictly private. That's the case with token-manager — you want your clients to be able to log out (delete token) but you don't want them to be able to add new token without going through one of auth services. This could be easily handled by a proxy server (like HAProxy or nginx) but sometimes you may want to do more complex transformations (ex. change API). In this case you should write a proxy service. Session manager is an exemplary proxy service that changes API and hides internals.

> See the asynchronous HTTP stream communication.

3.2. Asynchronous HTTP stream — akka–http

Asynchronous communication, unlike synchronous, is not restricted by any timing signals. Asynchronous communication is usually implemented by message passing (vs request–response) — 'telling' (vs 'asking'). Asynchronous protocols usually scales better as communicating parties don't have to wait. However asynchronous message passing is unnatural, can get really complex thus it's very often complicated and hard to debug. Nonetheless, truly reactive systems should relay on asynchronous message–passing — as stated in Reactive Manifesto.

metrics-common

In the previous chapter you saw nice interface for reporting metrics. Let's see how it's built. metrics-common is a shared library that consists of:

- Data structures definition & JSON mappers
- Metrics trait that:

hooks into actor system of parent service,

establishes long–lived HTTP connection to metrics-collector using Akka streams,

reestablishes connection on failure,

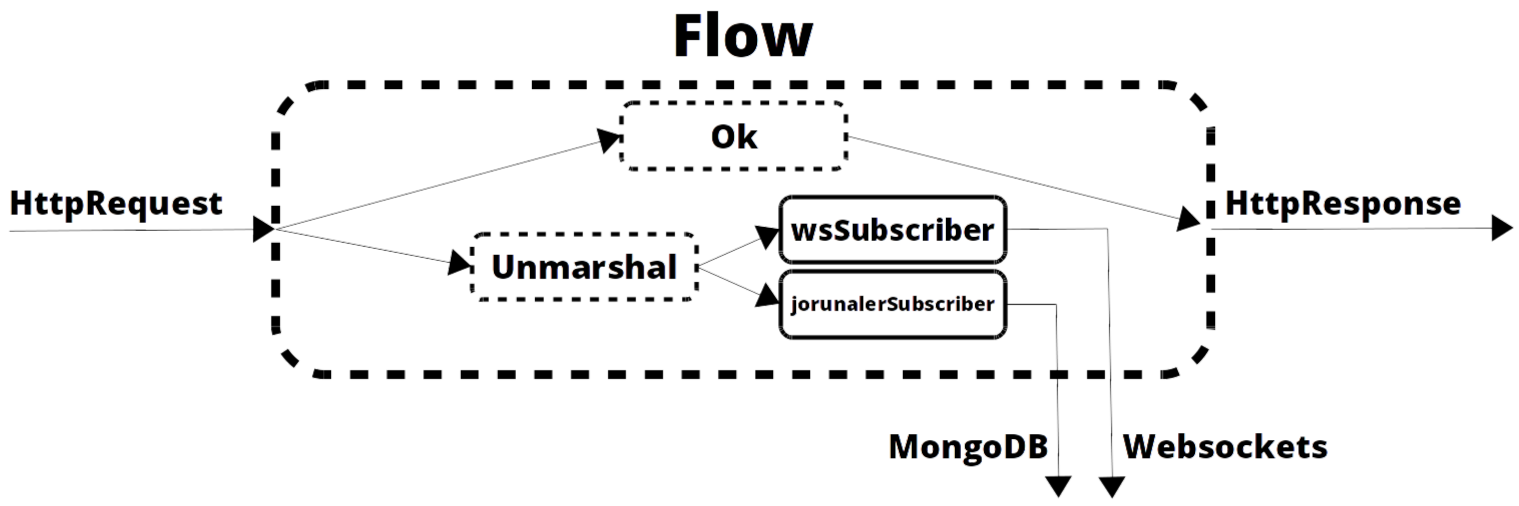
defines MetricsSupervisor actor that helps handling failures,

defines MetricsManager actor publisher that actually puts metric messages in the stream via a bounded buffer.

One may argue that this approach isn't asynchronous because we're using well–known HTTP requests. That's not true at all — we're using long–lived HTTP connection which provides us with an efficient stream and what's even more important, we're issuing requests without waiting for responses — we explicitly ignore them! This makes HTTP a one–way message stream and enables asynchronous processing. Another thing worth noting is fact we implemented metrics via a shared library that we declared a bad practise before because it introduces coupling. That's true but having nice and clean interface for reporting metrics simply outweighs the drawbacks. Still, if author of another microservice don't like that or don't use Scala, she can communicate with metrics-collector using HTTP and JSON, completely ignoring metrics-common library.

metrics-collector

metrics-collector is the receiving end of metrics subsystem. It's a Play app that receives metrics, stores them in a database and presents them to system administrators via websockets. Websockets part is really straightforward — it passes received metrics to websocket. The receiving endpoint is much more interesting. It's of course built using Akka HTTP but with customized flow instead of typical routing. The flow starts with broadcasting to `requestResponseFlow` and `requestMetricFlow`. `requestResponseFlow` simply maps every request to 200 HTTP response. `requestMetricFlow` turns requests into Metrics or in case of error to 'empty' element.`requestResponseFlow` is the actual output of the main flow while `requestMetricFlow` is being broadcasted to `wsSubscriber`and `journalerSubscriber`. `journalerSubscriber` is an actor subscriber that saves received metrics to MongoDB (ex. for further analysis). `wsSubscriber` is another actor subscriber that broadcasts received metrics to all connected websockets via Akka router.



If you want to have a closer look at this schema go here.

See the asynchronous HTTP websocket communication.

3.3. Asynchronous HTTP — websockets in Play

Another way to provide asynchronicity in the world of webservers are websockets. Websockets are similar to TCP sockets but additionally they provide minimal framing. This makes them ideal for asynchronous message passing. You've seen them in action in metrics-collector but this time we'll analyse more complex service.

btc-ws

btc-ws is a façade service for business part of our application. It allows users, after authentication, to manage subscriptions for BTC market events and receive alerts. While one could argue that user's actions can be handled synchronously, market events are asynchronous and should be handled like that. Websockets are perfect fit for such case — otherwise we would have to use HTTP polling which is inefficient. First thing to notice in the btc-ws code is a long block of mappings needed for a websocket message — scala object translations. Websocket initialization code is really straightforward — it retrieves user's identity based on presented token and opens websocket handled by `WebSocketHandler` actor. You'll learn what happens there in the next chapter.

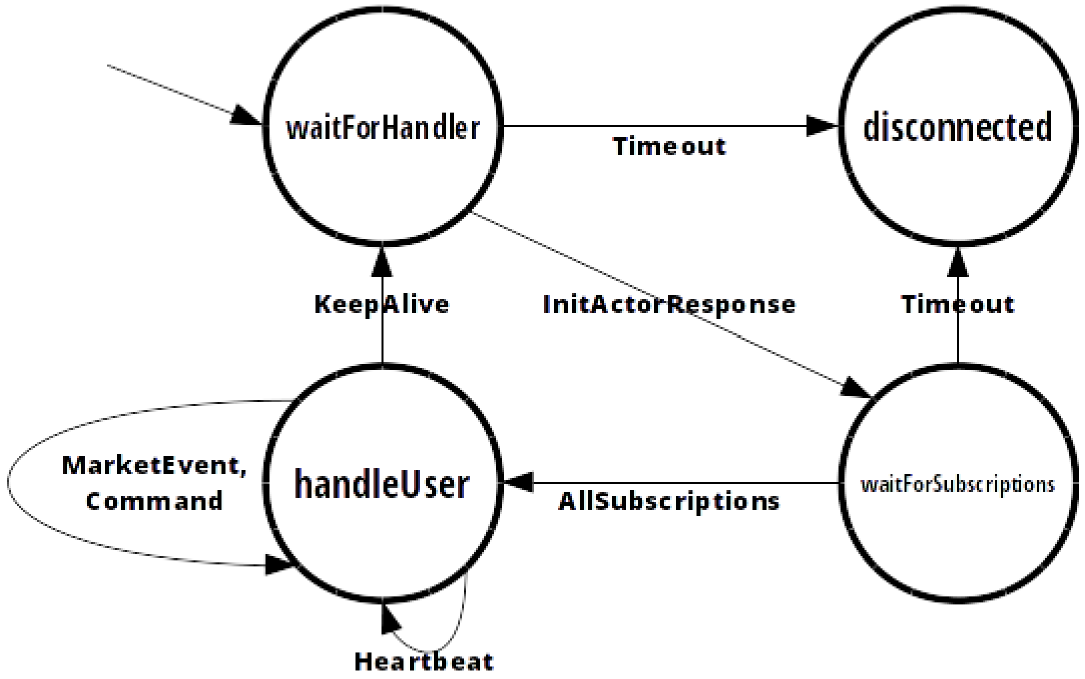
See the asynchronous messaging with Akka actors.

3.4. Asynchronous messaging — Akka actors

The 'go–to' tool when it comes to asynchronous message passing in Scala world is of course Akka. It has great capabilities and convenient interface but it comes with a cost. If you chose Akka as a communication protocol for a significant part of your project you're losing many of the benefits of polyglot approach. First of all, if you want to interoperate with Akka, you have to be on JVM and preferably use Java or Scala. Your code may also become more tightly coupled — Akka encourages usage of shared libraries and data structures. As a result, in certain cases it might be better to consider using lightweight message queues such as RabbitMQ or Kafka to avoid aforementioned drawbacks but if you're sure you won't be leaving JVM anytime soon, Akka is definitely the best choice. That was also our decision in the fully asynchronous part of our system.

btc-ws continued

`WebSocketHandler` actor is modelled as a simple state machine to handle all the possible failures. Let's see a state diagram.



When websocket connects and actor starts it sends a request to btc-users supervisor to get a remote command handler (`ActorRef`) (`waitForHandler` state), in case of failure it simply disconnects websocket — there's nothing we can do more here. After successful handler acquiring (`waitForSubscriptions` state), websocket actor requests a list of existing subscriptions for user — handler failure is the same as before. List of subscriptions causes websocket actor to enter operational handling of commands (which are routed to handler) and market events from handler (which are routed to websocket) — `handleUser` state; in case of timeout we assume something bad happened to handler so we switch back to `waitForHandler` state. Notice how clean protocol we've got — btc-ws microservice doesn't know any implementation details of btc-users — it just sends messages with clear semantics.

btc-users

btc-users is a microservice completely based on Akka. It consists of three actors types: `UsersManager`, `UserHandler` and`DataFetcher`. `UsersManager` plays a role of supervisor. It responds to requests from btc-ws to create a handler (`UserHandler`actor) for user with given id. `UserHandler` is where all the heavy lifting happens. It's a persistent actor that processes subscription requests and issues alarms based on ticker from BTC market. First and foremost we once again leveraged the polyglot persistence approach — we used Akka persistence to persist subscription settings. Subscribe/unsubscribe actions are a perfect cases for event sourcing and that's exactly how we implemented it — see `receiveCommand` and `receiveRecover`. Besides handling subscribe/unsubscribe requests, `UserHandler` responses to `QuerySubscriptions` and broadcasts market alarms. `UserHandler`actor manages its lifetime similar to how btc-ws does — by heartbeats and timeouts. `DataFetcher` is a very simple actor that every few seconds fetches BTC ticker and broadcasts it to all `UserHandler` actors via Akka router. That's it — that's how subscriptions are managed and market alarms are issued.

See the tutorial summary.

4. Summary

During our tour of microservices we hopefully showed the full power that comes with mixing different approaches, techniques and tools. Scala's toolbelt (and especially Akka and Play) is, without a question, ready for building reactive microservice–based distributed systems. However before migrating all your project to MSA make sure you deeply understand all dilemmas and problems of microservices, particularly ones we had to omit to keep this activator template concise like: eventual consistency, testing, operations & deployment, contract managing, versioning, monitoring, logging & debugging and security. Good luck, have fun and let us know about your adventures on the microservice way!