[axonlogo] | *axoniq.png*

Recipe

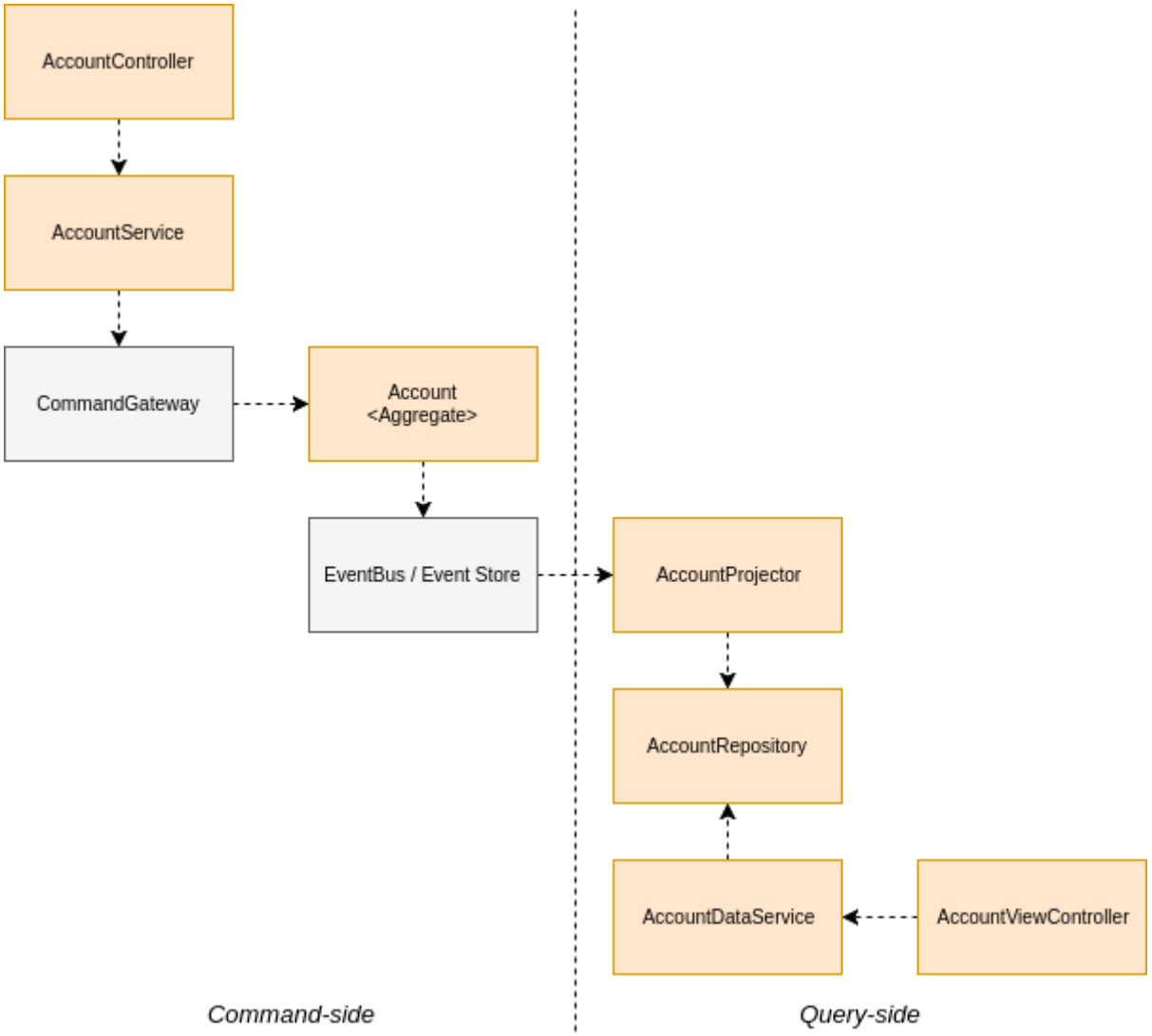
Axon Framework in a Spring Boot application

This recipe gives you the basic steps for creating an Axon Framework based application with the use of Spring Boot. The Axon Framework is not tied to Spring. You can find the recipe on how to create a basic Axon application without the use of Spring here.

This recipe will focus on the implementation of an Axon application and not the concepts were the Axon Framework is based on. You can find the in dept information on the Axon Framework in the Axon Framework reference guide

All the recipes are based on one domain, a bank. Axon Framework with CQRS and eventsourcing applies very well to the domain of a bank. A person wants to open a bank account to deposit and withdraw money from. Over time, the customer wants to see the current state of the account and see which transactions have taken place to get to the current amount. The events that have been applied over time are the basis of the transaction log and the current amount is representation of the state. Next to depositing and withdrawing money, the bank wants to know all changes that have taken place to an account for auditing purposes. The last version of its details is shown to the accountholder, but the bank might want to see the change log, i.e. for auditing purposes, instead of only the last state.

This recipe will lead you through the first step of creating a bank account and depositing and withdrawing money. The following diagram gives an overview of the classes we will create in the application. The command-side represents the classes related to creating and writing the events. The query-side represents the handling of the events and requests for the projection of the data.



General

- Cooks in: 30 minutes
- Difficulty: easy

Ingredients

- **Dependencies**

Dependency	Version
Axon Framework	3.0.X
Spring Boot starter for Axon Framework	3.0.X
Spring Boot	1.5.X.RELEASE
Kotlin	1.1.X

- **Axon Framework components**

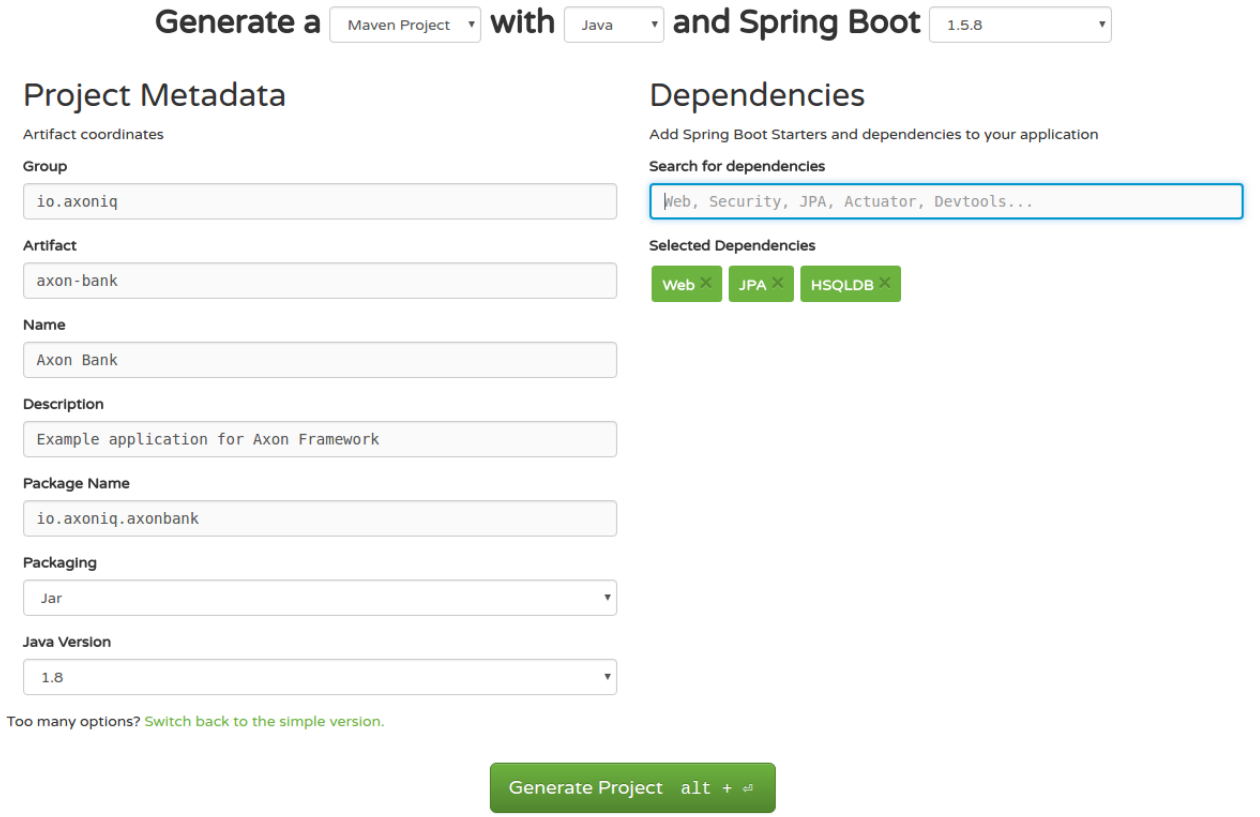
Aggregate

Command handlers
Event handlers
Event Sourcing handlers

Method

1 Set up the Spring Boot application

For the creation of a simple Spring Boot application you can go to start.spring.io. You can configure the application metadata as preferred. In this example we added `Web`, `JPA` and `HSQLDB` as dependencies. After configuring everything you can download the Spring Boot application and load as a project in your preferred IDE.



2 Add dependencies

Navigate to the pom.xml. There will be already a set of Spring Boot related dependencies. Add the Axon Framework Spring Boot starter dependency to the list.

```
<dependency>
    <groupId>org.axonframework</groupId>
    <artifactId>axon-spring-boot-starter</artifactId>
    <version>3.0.6</version>
</dependency>
```

For the value objects we will make use of Kotlin. To use Kotlin, add the following also to the list of dependencies.

```
<dependency>
    <groupId>org.jetbrains.kotlin</groupId>
    <artifactId>kotlin-stdlib</artifactId>
    <version>1.1.51</version>
</dependency>
```

For building the application you will need to two following three plugins to the pom.xml

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
            <version>${kotlin.version}</version>
            <executions>
                <execution>
                    <id>compile</id>
                    <phase>compile</phase>
                    <goals>
                        <goal>compile</goal>
                    </goals>
                </execution>
                <execution>
                    <id>test-compile</id>
                    <phase>test-compile</phase>
                    <goals>
                        <goal>test-compile</goal>
                    </goals>
                </execution>
            </executions>
            <configuration>
                <jvmTarget>1.8</jvmTarget>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <executions>
                <execution>
                    <id>default-compile</id>
                    <phase>none</phase>
                </execution>
                <execution>
                    <id>default-testCompile</id>
                    <phase>none</phase>
```

```
                </execution>
                <execution>
                    <id>java-compile</id>
                    <phase>compile</phase>
                    <goals>
                        <goal>compile</goal>
                    </goals>
                </execution>
                <execution>
                    <id>java-test-compile</id>
                    <phase>test-compile</phase>
                    <goals>
                        <goal>testCompile</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

3 Start the application

You should be able to run the Axon Spring Boot application now without any issues. If not, please check your configuration and Spring documentation before going to the next steps.

4 Command-side implementation

We will need a couple of endpoints, one endpoint to create a bank account and two endpoints for depositing and withdrawing money.

```
@RestController
public class AccountController {

    private static final Logger log = LoggerFactory.getLogger(AccountController.class);

    private final AccountService accountService;

    @Autowired
    public AccountController(AccountService accountService) {
        this.accountService = accountService;
    }

    @PostMapping("/account")
    public ResponseEntity createBankAccount(@RequestBody String name) {
        log.info("Request to create account for: {}", name);

        UUID accountId = accountService.createBankAccount(name);

        return new ResponseEntity<>(accountId, HttpStatus.CREATED);
    }

    @PutMapping("/account/{accountId}/deposit/{amount}")
    public ResponseEntity depositMoney(@PathVariable UUID accountId, @PathVariable Double amount) {
        log.info("Request to withdraw {} dollar from account {} ", amount, accountId);

        accountService.depositMoney(accountId, amount);

        return new ResponseEntity(HttpStatus.OK);
    }

    @PutMapping("/account/{accountId}/withdraw/{amount}")
    public ResponseEntity withdrawMoney(@PathVariable UUID accountId, @PathVariable Double amount) {
        log.info("Request to withdraw {} dollar from account {} ", amount, accountId);

        accountService.withdrawMoney(accountId, amount);

        return new ResponseEntity(HttpStatus.OK);
    }

}
```

We need three commands for our logic of creating an account, depositing and withdrawing money. We will create the `CreateAccountCommand`. `DepositMoneyCommand` and `WithdrawMoneyCommand`. Next to the id of the account the name of the accountholder is saved. The `@TargetAggregateIdentifier` annotation is required for command handling in the aggregate. By using this annotation, Axon knows which aggregate to target when handling the command.

```
data class CreateAccountCommand(
        @TargetAggregateIdentifier val accountId: UUID,
        val name: String?
)

data class DepositMoneyCommand(
        @TargetAggregateIdentifier val accountId: UUID,
        val amount: Double
)

data class WithdrawMoneyCommand(
        @TargetAggregateIdentifier val accountId: UUID,
        val amount: Double
)
```

The `AccountController` is actually a simple controller which is just receiving data via the endpoints and passing it through to the service. Our `AccountService` will do the logic of validating the input and sending the commands. Validation should only be on level of input parameters being valid. Logic on whether money can be withdrawn from the account will be done in the command-handling model, the `Account` aggregate.

```
@Service
public class AccountService {

    private final CommandGateway commandGateway;

    @Autowired
    public AccountService(CommandGateway commandGateway) {
        this.commandGateway = commandGateway;
    }

    public UUID createBankAccount(String name) {
        assertNotNull(name, "The name of the account holder should not be null");

        UUID accountId = UUID.randomUUID();

        CreateAccountCommand createAccountCommand = new CreateAccountCommand(accountId, name);
        commandGateway.send(createAccountCommand);

        return accountId;
    }

    public void depositMoney(UUID accountId, Double amount) {
        commandGateway.send(new DepositMoneyCommand(accountId, amount));
    }

    public void withdrawMoney(UUID accountId, Double amount) {
        commandGateway.send(new WithdrawMoneyCommand(accountId, amount));
    }

}
```

The events will be applied in the `Account` aggregate. For the domain of the bank, the events will be almost a one to one mapping of the commands. Although, in some cases the aggregate will handle a command and apply multiple events or the event might contain calculated data. For example, in the case of the bank application the balance could be included in the event.

```
data class AccountCreatedEvent(
        val accountId: UUID,
        val name: String?
)

data class MoneyDepositedEvent(
        val accountId: UUID,
        val amount: Double
)

data class MoneyWithdrawnEvent(
        val accountId: UUID,
        val amount: Double
)
```

The `Account` aggregate holds the state of the bank account. Commands are handled and when a change should be made to the state events will be applied. Important to mention is that the events will change the state of the aggregate. The command will only use the state of the aggregate to determine whether an event can be applied.

The aggregate will start by first handling the command that does the creation of an object. In the case of the bank account the `CreateAccountCommand`. This `command` should be handled in the constructor of the class.

```
// Required for Axon to create the aggregate [requires more explanation]
public Account() {}

@CommandHandler
public Account(CreateAccountCommand command) {
    apply(new AccountCreatedEvent(command.getAccountId(), command.getName()));
}
```

The `CreateAccountCommand` does not require any validation for now. The `AccountCreatedEvent` event can be applied directly. To initialize the state of the `Account` aggregate with the id and a default balance, an `EventSourcingHandler` is used. The events are handled the next time the aggregate is retrieved for handling a new command. The state of the aggregate needs to be build up based on the past events.

```
@AggregateIdentifier
private UUID accountId;

private Double balance;

@EventSourcingHandler
protected void on(AccountCreatedEvent event) {
    this.accountId = event.getAccountId();
    this.balance = 0.0;
}
```

The other commands will be handled in methods in the aggregate. The full aggregate with handling all commands and events will look like shown in the following snippet.

```
@Aggregate
public class Account {

    @AggregateIdentifier
    private UUID accountId;

    private Double balance;

    // Required for Axon to create the aggregate
    public Account() {
    }

    @CommandHandler
    public Account(CreateAccountCommand command) {
        apply(new AccountCreatedEvent(command.getAccountId(), command.getName()));
    }

    @CommandHandler
    public void handle(DepositMoneyCommand command) {
        apply(new MoneyDepositedEvent(command.getAccountId(), command.getAmount()));
    }

    @CommandHandler
    public void handle(WithdrawMoneyCommand command) {
        if (balance - command.getAmount() >= 0) {
            apply(new MoneyWithdrawnEvent(command.getAccountId(), command.getAmount()));
        }

        // todo add exception handling
    }

    @EventSourcingHandler
    protected void on(AccountCreatedEvent event) {
        this.accountId = event.getAccountId();
        this.balance = 0.0;
    }

    @EventSourcingHandler
    protected void on(MoneyDepositedEvent event) {
        this.balance = balance + event.getAmount();
    }

    @EventSourcingHandler
    protected void on(MoneyWithdrawnEvent event) {
        this.balance = balance - event.getAmount();
    }

}
```

5 Query-side implementation

The `Account` aggregate keeps state of the account based on the events. So, by using the event store it is possible to see the steps taken to get to the current balance. Although, replaying all the events every time we want to show the balance is a bit too much effort. Therefore, we have the query side to listen to the events and create a temporary projection of the state. Each time an event will come in, the projection will be updated. Every view that needs the data can then just query the system and will receive the current state.

The first step to accomplish this is creating the projector and using the `@EventHandler` annotation. In the example the projection is written to a database table, but many other options are possible, like publishing the latest projection to a RabbitMQ queue or sending out an email.

When the `AccountCreatedEvent` is handled the view is created, the other events will update the view over time.

```
@Service
public static class AccountProjector {

    private final AccountRepository repository;

    @Autowired
    public AccountProjector(AccountRepository repository) {
        this.repository = repository;
    }

    @EventHandler
    public void on(AccountCreatedEvent event) {
        AccountView accountView =
                AccountView.builder()
                           .accountId(event.getAccountId())
                           .name(event.getName())
                           .build();

        repository.save(accountView);
    }

    @EventHandler
    public void on(MoneyDepositedEvent event) {
        UUID accountId = event.getAccountId();
        AccountView accountView = repository.getOne(accountId);

        double newBalance = accountView.getBalance() + event.getAmount();

        AccountView updatedView = AccountView.builder()
                                             .copyOf(accountView)
                                             .balance(newBalance)
                                             .build();

        repository.save(updatedView);
    }
```

```
    @EventHandler
    public void on(MoneyWithdrawnEvent event) {
        UUID accountId = event.getAccountId();
        AccountView accountView = repository.getOne(accountId);

        double newBalance = accountView.getBalance() - event.getAmount();

        AccountView updatedView = AccountView.builder()
                                             .copyOf(accountView)
                                             .balance(newBalance)
                                             .build();

        repository.save(updatedView);
    }

}
```

To keep the code concise in this recipe, we save all fields on database entity level. Preferred is an object in between that maps the state representation to a database object. For example `AccountView` and `AccountViewDao`.

Due to running the embedded database in this code exmampe, we need to add a simple constructor to the database object: `constructor() : this(UUID.randomUUID(), null, 0.0) {}`. For now, we just add some dummy data in here.

```
@Table(name = "account")
@Entity(name = "account")
data class AccountView(
        @Id val accountId: UUID,
        val name: String?,
        val balance: Double
) {

    // Required for running embedded db [needs more explanation]
    constructor() : this(UUID.randomUUID(), null, 0.0) {}

    class Builder {
        private lateinit var accountId: UUID
        private var name: String? = null
        private var balance: Double = 0.0

        fun accountId(v: UUID) = apply { accountId = v }
        fun name(v: String?) = apply { name = v }
        fun balance(v: Double) = apply { balance = v }

        fun copyOf(v: AccountView) = apply {
            accountId = v.accountId
            name = v.name
            balance = v.balance
        }

        fun build() = AccountView(accountId, name, balance)
    }

    companion object {
        @JvmStatic fun builder() = Builder()
    }

}
```

Due to the use of JPA we only have to create an `AccountRepository` that extends from the `JpaRepository`. JPA will take care of creating methods as `save()`, `findOne()` and `findAll()`

```
public interface AccountRepository extends JpaRepository<AccountView, UUID> {}
```

To query a specific account or query all the accounts, we create two endpoints. The controller requests the `AccountDataService` for information on one or more accounts. In this case the calls are simple and straightforward, but there could be cases where additional information should be added to the view (i.e. from other aggregates) or the projection should be filtered depending on the request parameters.

```
@RestController
public static class AccountViewController {

    private static final Logger log = LoggerFactory.getLogger(AccountViewController.class);

    private final AccountDataService accountDataService;

    @Autowired
    public AccountViewController(AccountDataService accountDataService) {
        this.accountDataService = accountDataService;
    }

    @GetMapping("/account/{accountId}")
    public AccountView getAccountById(@PathVariable UUID accountId) {
        log.info("Request Account with id: {}", accountId);

        return accountDataService.getAccountById(accountId);
    }

    @GetMapping("/accounts")
    public List<AccountView> getAllAccounts() {
        log.info("Request all Accounts");

        return accountDataService.getAllAccounts();
    }

}
```

```
@Service
public static class AccountDataService {

    private final AccountRepository accountRepository;

    @Autowired
    public AccountDataService(AccountRepository accountRepository) {
        this.accountRepository = accountRepository;
    }

    public AccountView getAccountById(UUID accountId) {
        return accountRepository.findOne(accountId);
    }

    public List<AccountView> getAllAccounts() {
        return accountRepository.findAll();
    }

}
```

Other readings and recipes

By following this recipe, you should be able to run a simple application using Spring Boot and the Axon Framework. Of course, over time the bank will find out that there can be more than one type of bank account that can be created or not all values required for withdraw are in the event. In that case, we will need to change the events and maybe adjust the events. We start simple in this example application, just a basic set of events. But at the bottom you can find a reference to other recipes for cooking upcasters, event stores, command busses and more. Let's first start with this recipe.

- How to write an Aggregate test
- How to write an Upcaster
- How to add a Saga
- How to add an Entity under an Aggregate
- Axon Framework reference guide

For any question about the recipe, please contact

- Axon user group
- AxonIQ support