Music Event System

SWEN90007 Software Design and Architecture

Project: Part 2

Software Architecture Report (SAD)

TEAM NAME - **MusicBandTeam**

Team Members	Student ID
Bingqing Zhang	1377017
Xin Xiang	1294725
Dhakshayani Perumal	1184165
Vivien Guo	1173459

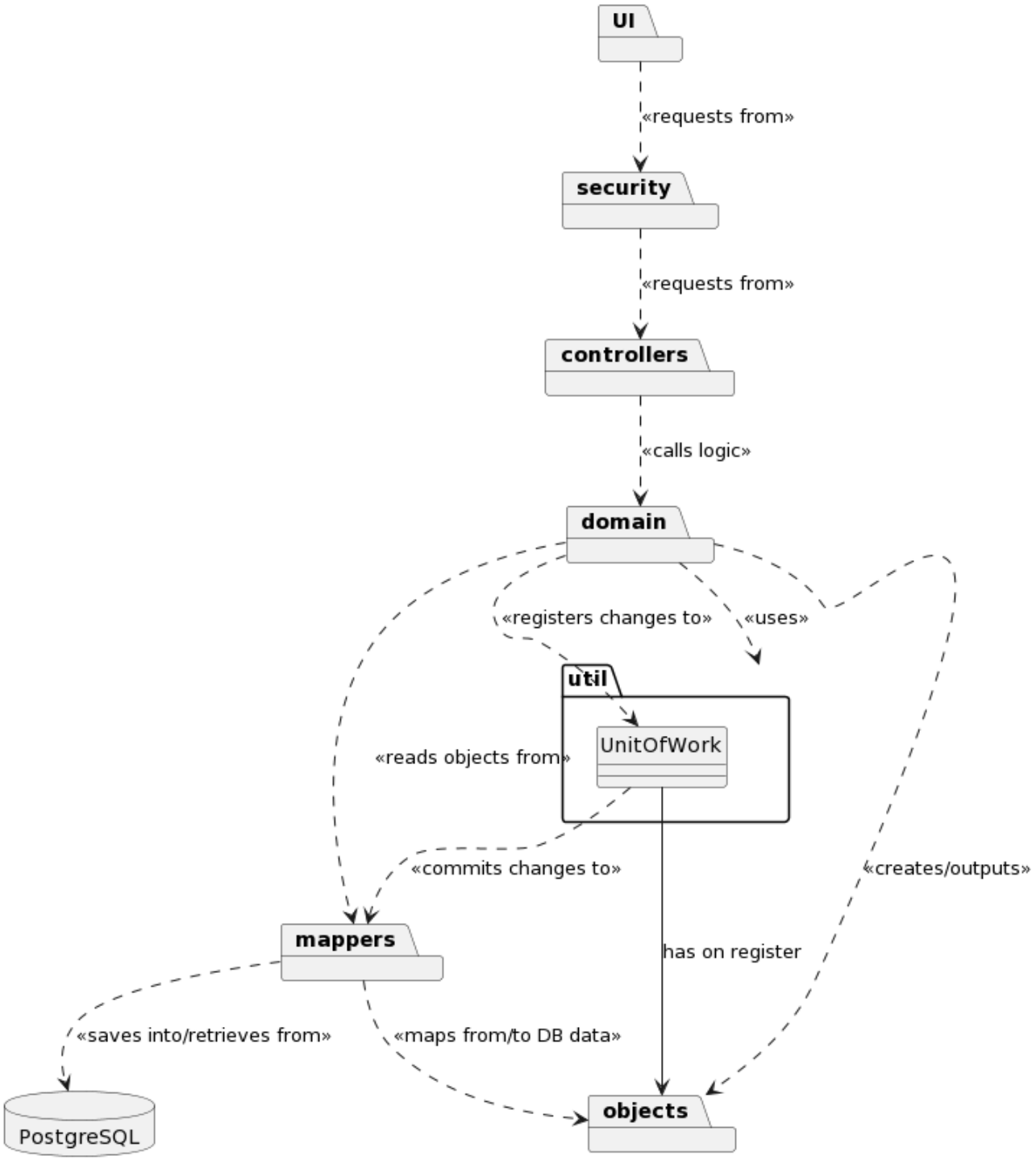
Class Diagram

For legibility, the following items are omitted:

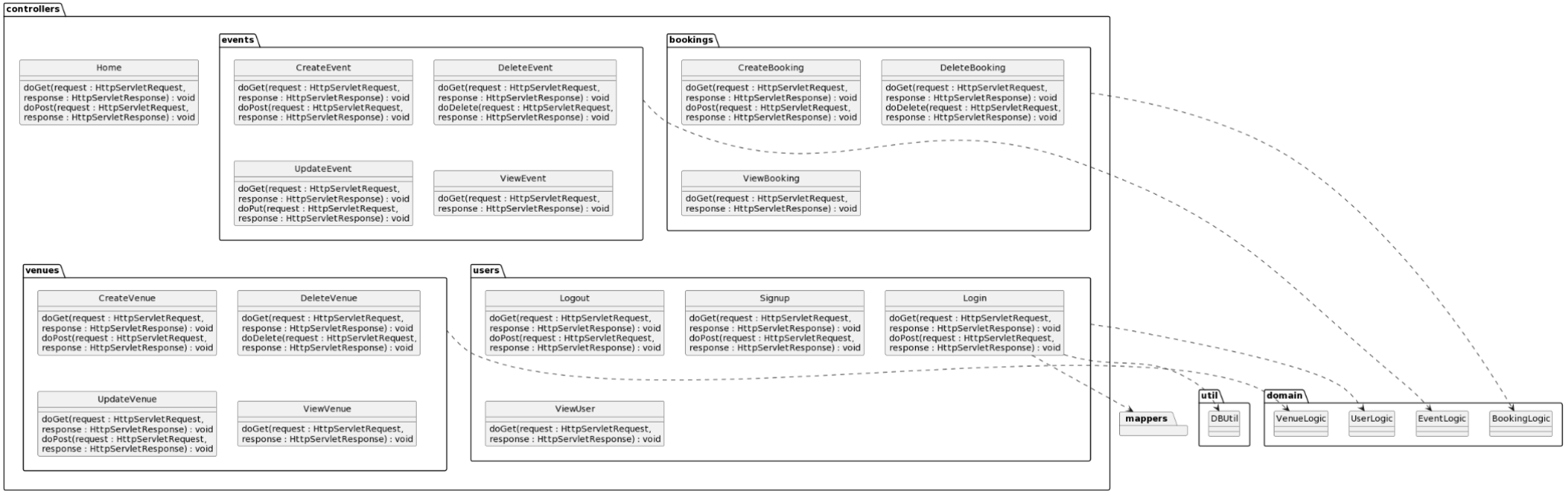
- Route methods (doGet, doPost etc.)
- Overridden methods in implementations & subclasses
- Trivial initializers, getters and setters
- Custom exceptions and any unused classes

The whole class diagram is too big to be viewed all at once, so we split it up into the following parts:

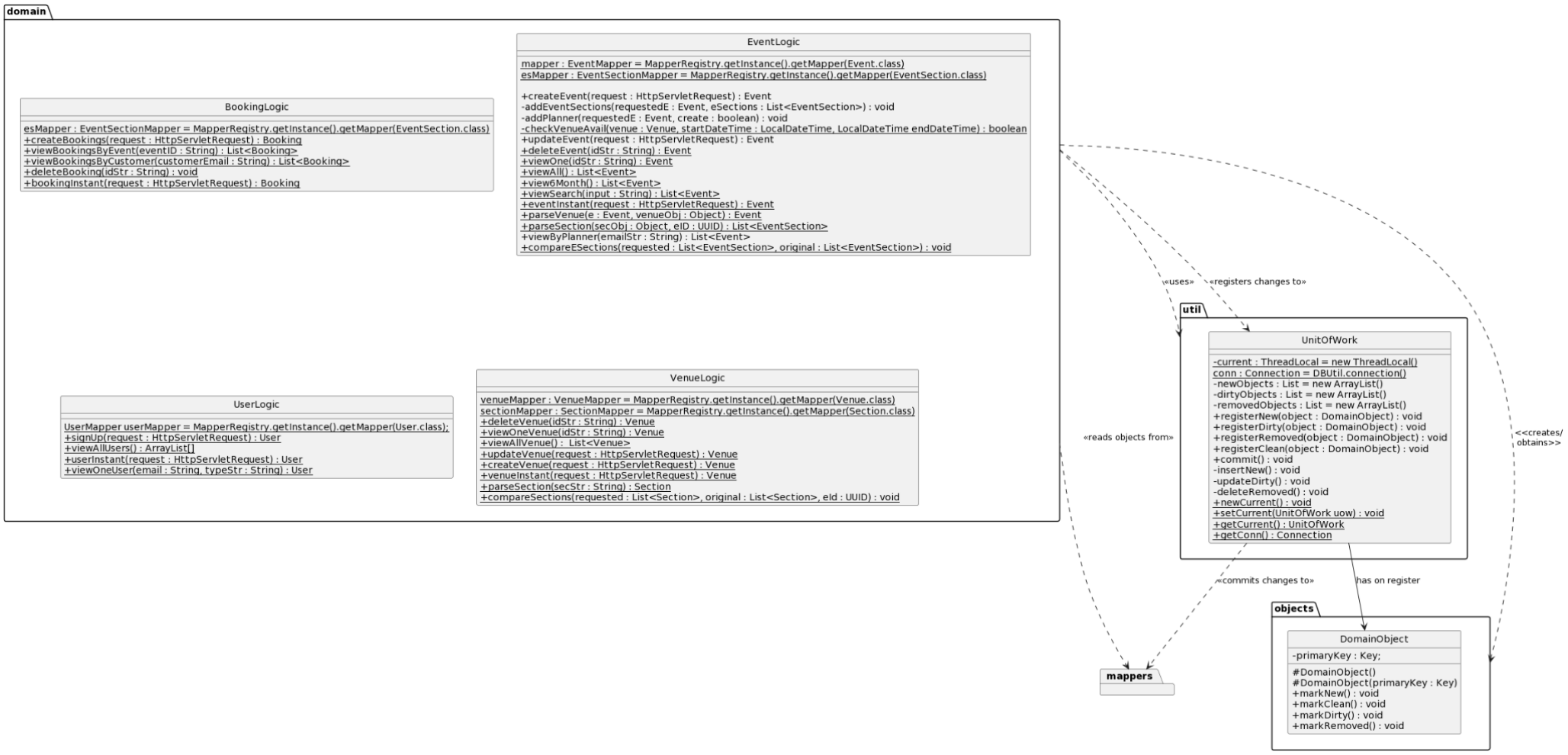
• Broad overview



- Controllers

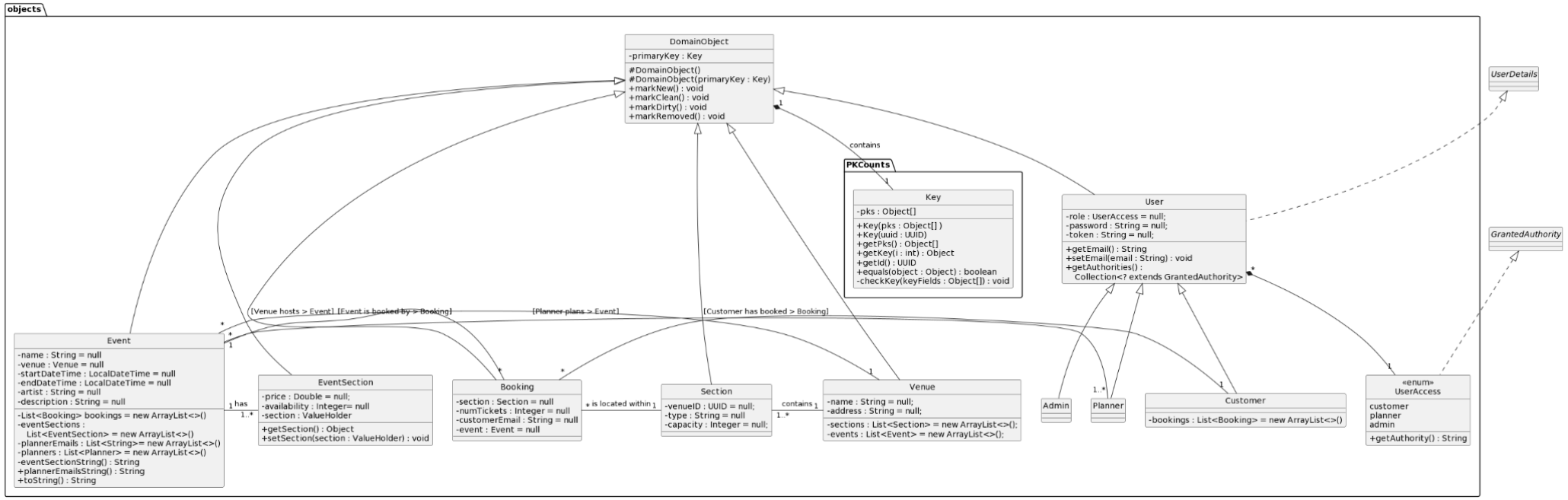


- Domain logic & Unit of work

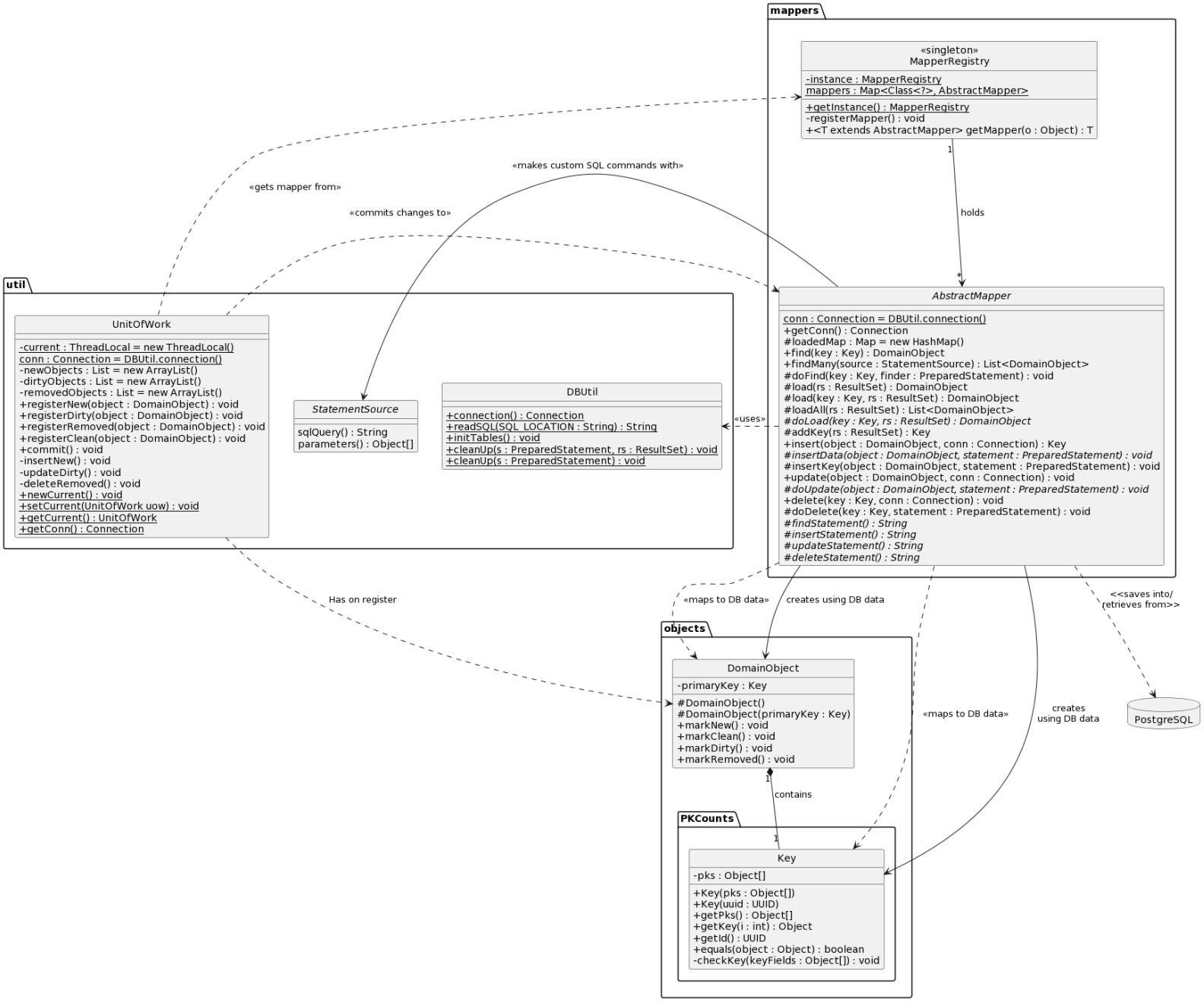


• Domain objects

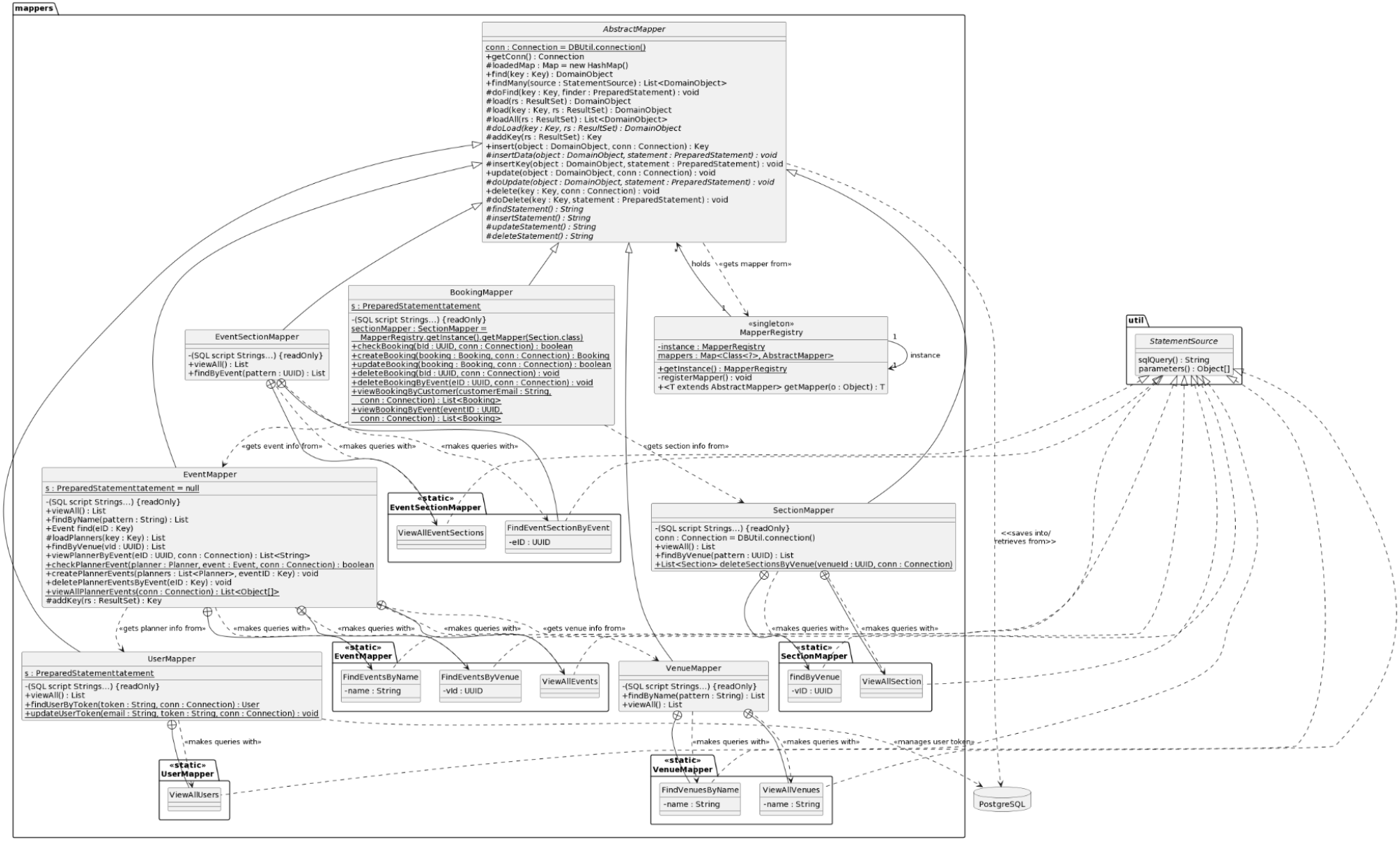
Note that UserDetails and GrantedAuthority refer to the Spring Security classes.



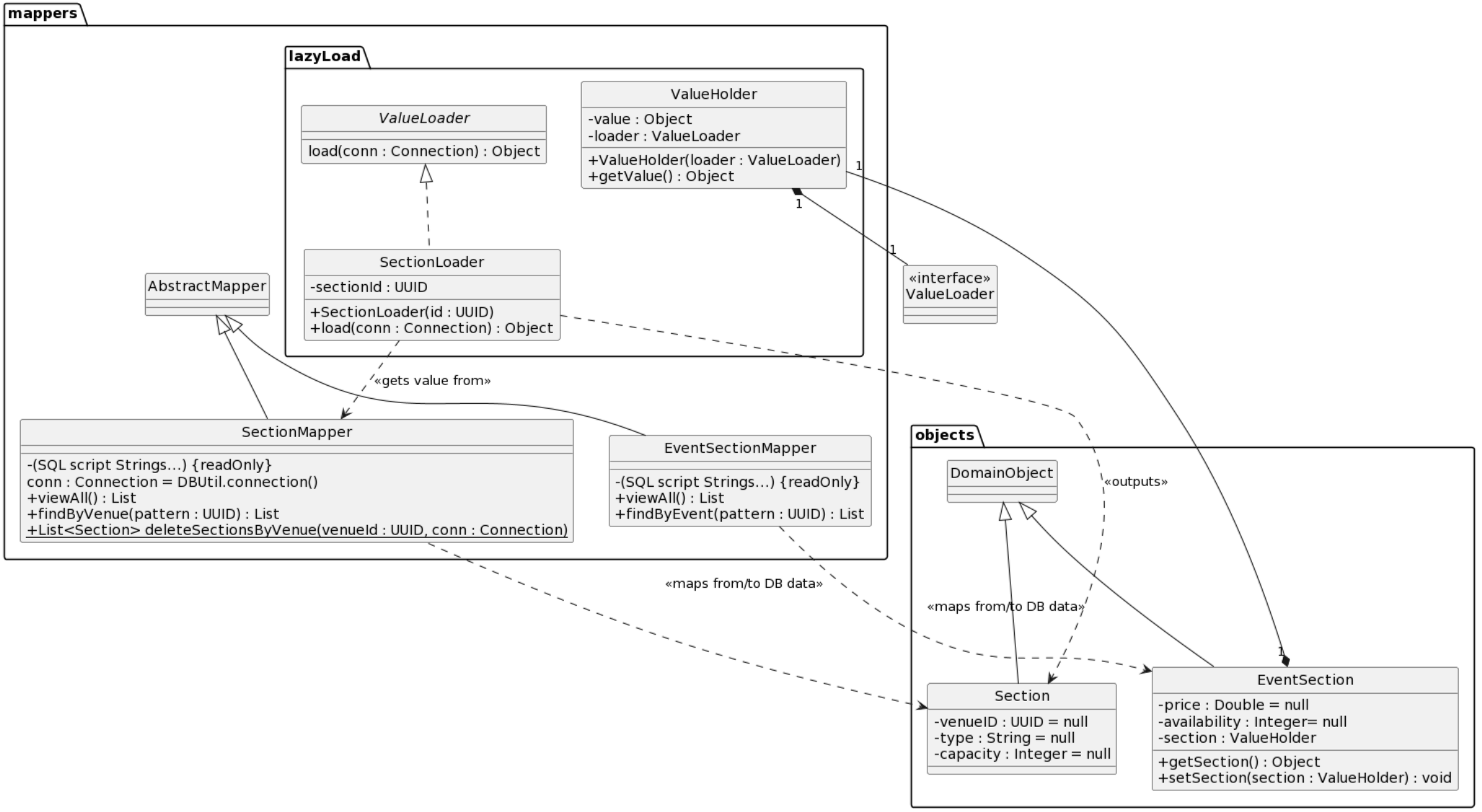
- Unit of work & abstract mapping process



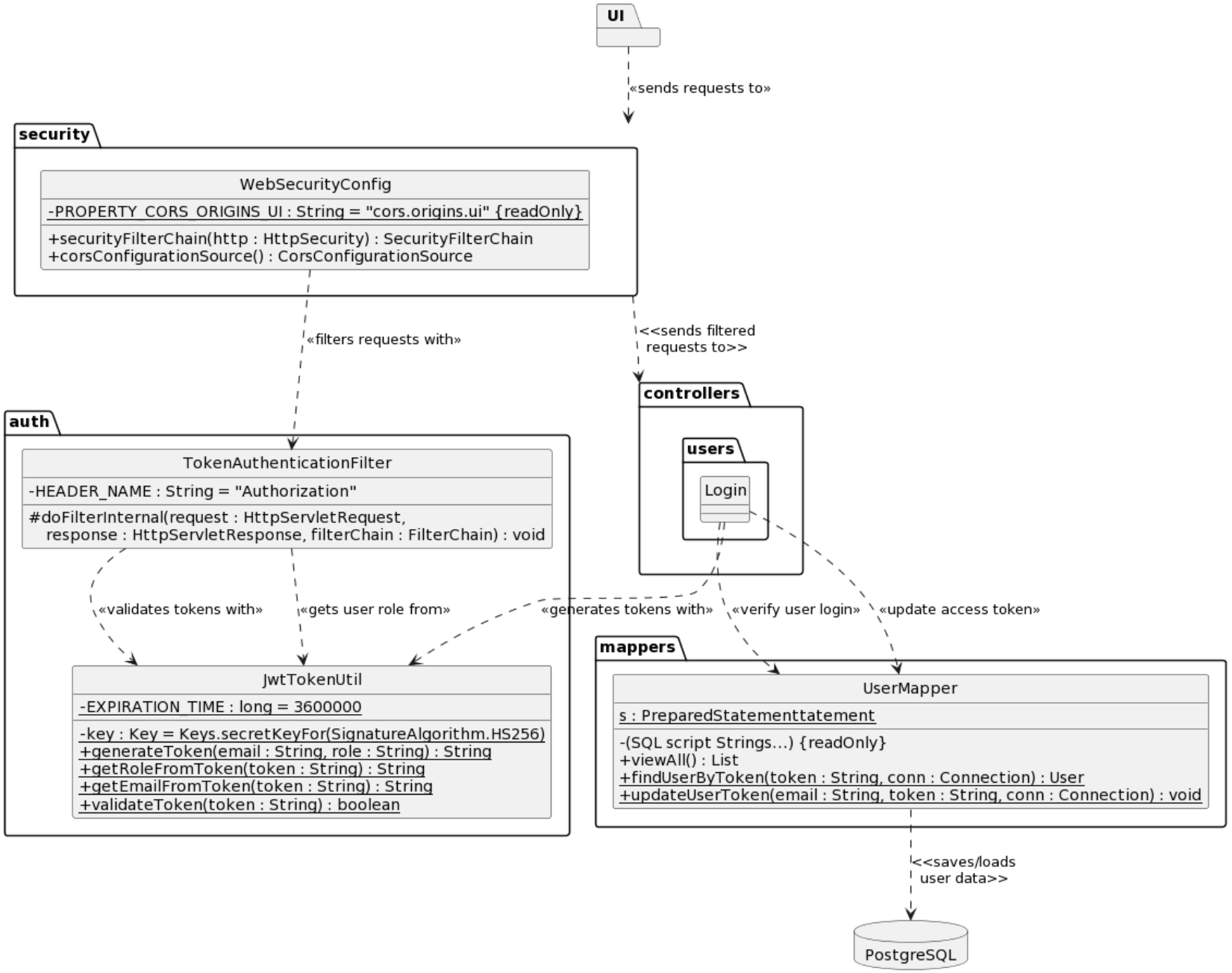
- Specific mappers



• Lazy loading & EventSectionMapper



- Authentication



Patterns used

• Domain model

We identify the following entities from the business domain: Venues, Sections, Events, Users, Admin, Planners, Customers, and Bookings. To model them, we create objects for each of them, so each object can keep track of its own data. For ease of use we also created the EventSection object, since the relationship between Events and Sections carry additional per-section-per-event data such as pricing and availability.

To increase abstraction, we created the Key class for each object class to use, as a way of easily working with them regardless of whether their primary keys are simple or compound (e.g. Venue with one field as primary key, EventSection with two fields as primary key.) This will be elaborated further in the Identity Field section.

We also created the general DomainObject superclass, which all object classes within the business domain extend from. These objects come with the ability to get and set their primary key (of type Key), as well as methods to register themselves onto the UnitOfWork.

(See this pattern in action in the Unit of Work section.)

When the backend logic receives object data from the frontend, it first parses the raw data into a DomainObject, then works together with the Data Mapper pattern below:

• Data mapper

To decrease coupling between our domain model and the database itself, we implemented data mappers for each object in the business domain. As a means of abstraction we first create an AbstractMapper, which deals with linking DomainObjects to the database. Specifically, it performs the following:

- CREATE: Given a DomainObject (insert), insert its data into a PreparedStatement, which is then executed. Afterwards, it returns a key to the domain logic for transmission to frontend.
- READ: Given a single Key (findOne) or a StatementSource* (findMany), find the corresponding rows from the database and load them into DomainObjects.
- UPDATE: Given a DomainObject (update), update its own relevant table.
- DELETE: Given a DomainObject (delete), delete the relevant item in the table. Note that by default it only requires for there to be a UUID stored within the DomainObject's Key

*StatementSource: This class has been created to support findMany operations. Since the business logic is not only restricted to retrieving all items/retrieving items by a simple primary key, abstracting the Statement usage helps with performing various READ operations (e.g. "`SELECT * FROM Events WHERE name = ?;`" for searching events by name, "`SELECT * FROM events;`" for retrieving all events).

All object mappers extend this AbstractMapper and, depending on their relevant fields, implement the helper functions for the key operations differently. By extending this parent class, we also enable the Unit of Work pattern to call our mappers during commits.

(See this pattern in action in the Unit of Work section.)

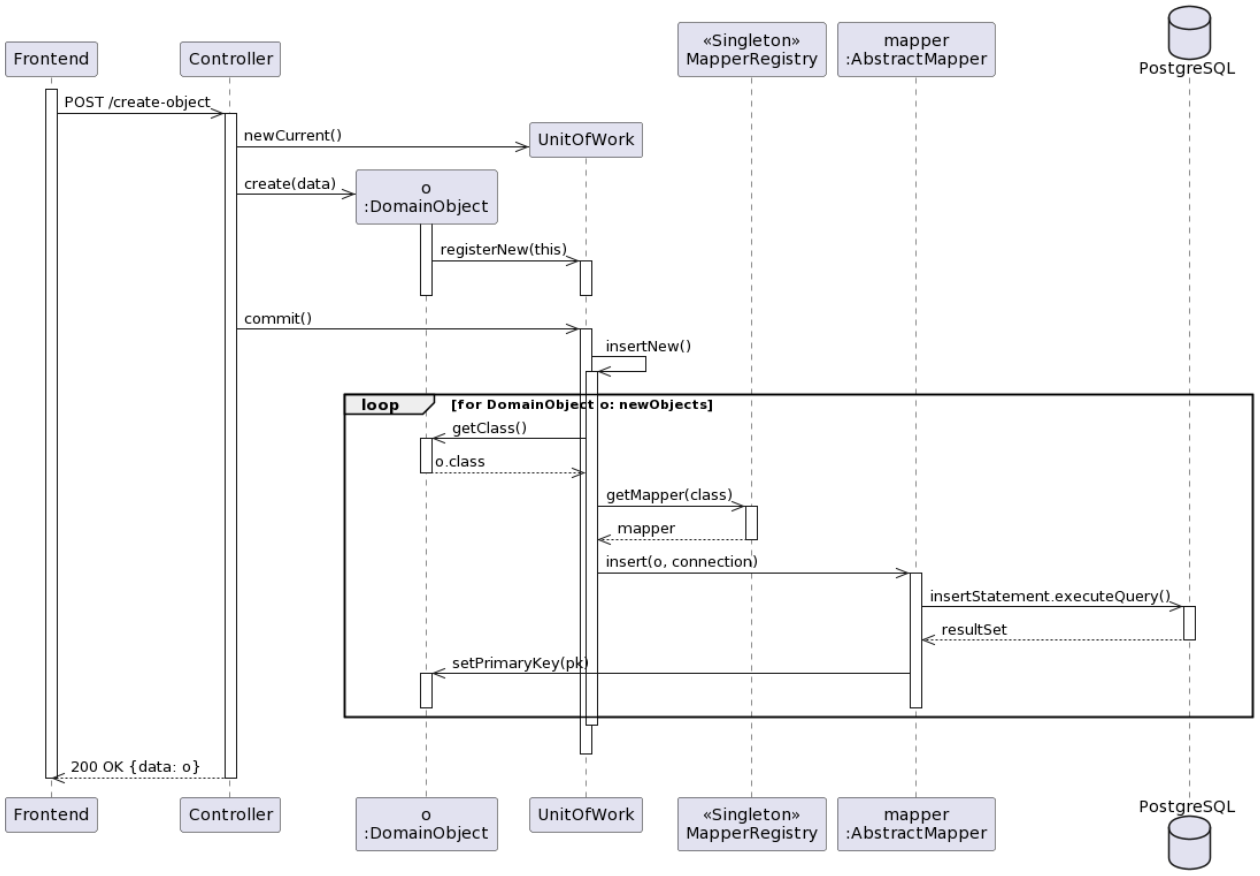
• Unit of work

Each time that our system runs a transaction, any relevant objects are stored in the UnitOfWork class. As noted in the Domain Mapper section, all of our objects extend from the class DomainObject, which allows one UnitOfWork class to store them all.

We implemented object registration on the DomainModel to support this, so that this process is automated upon object CRUD operations, by the object themselves. This removes the onus from the backend controllers and ensures that when developing possible logic extensions we will not forget to register objects onto the UnitOfWork. To enable this object registration, we also implemented the UnitOfWork as a singleton, so that all objects are able to access it during transactions.

At the end of each transaction, we call the commit method on the UnitOfWork, which will perform the following: create new objects, update dirty objects, delete removed objects. To generalise this process, for each relevant DomainObject we look up its corresponding Mapper through the MapperRegistry*, then call its insert/update/delete method.

*MapperRegistry: This class has been created to support the UnitOfWork. By storing a `Map<Class<?>, AbstractMapper>`, we enable the UnitOfWork to easily call a DomainObject's Mapper through its subclass.



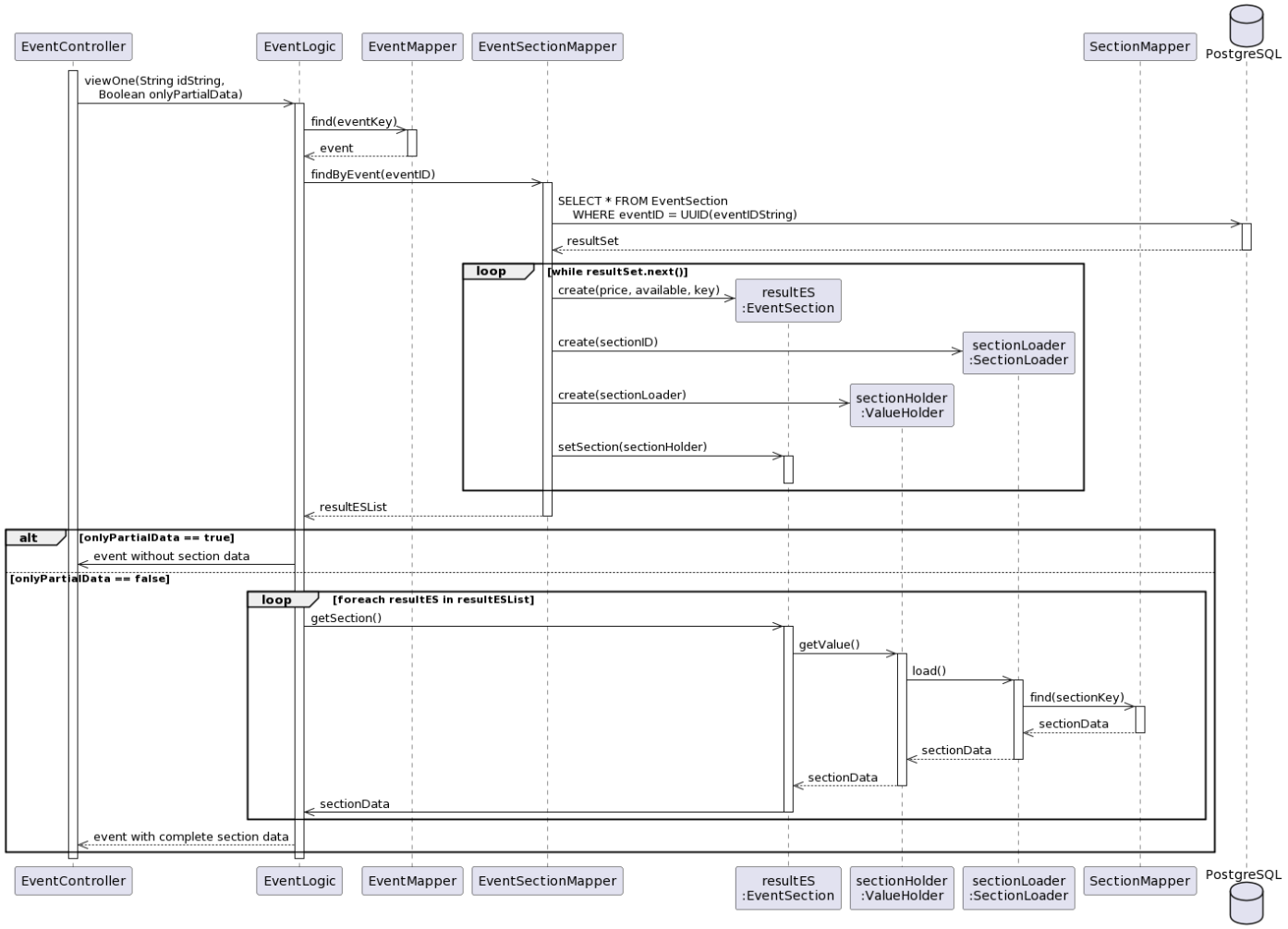
Our Domain Model, Data Mapper and Unit of Work acting together to process an abstract creation request from the frontend. Note that some items and steps (e.g. Controller, Singleton instance creation, UnitOfWork updating other changed data) have been simplified/omitted for legibility.

• Lazy load

In our application we implement lazy loading through the value holder pattern. Both the rationale for implementing lazy load in general and for choosing the value holder pattern are documented in the Design Rationale section.

We implemented the lazy load pattern for the Section data within an EventSection, as often it is unnecessary for it to be loaded when Users are requesting event data. When we request an Event from the database, we first retrieve a partially loaded version of it from the mapper (i.e. an Event without its Section type and capacities), with a ValueHolder instead of an actual Section. This ValueHolder comes with a SectionLoader (which implements the ValueHolder.ValueLoader) class, so it is able to query the database for its own data if necessary.

It is then up to our Logic classes to determine if the unloaded information is necessary or not. For example, the EventLogic would call the EventValueHolder (EventVH's) method for retrieving EventSection data in the use case of Planner Edit Event, while in the Customer View Events use case it would return the Events to the frontend right away, without making an extra call to the Sections table.



Our lazy loader pattern at work for Sections within our EventSections.
Note that variables are renamed and mapper calls to the database by EventMapper and SectionMapper is omitted for clarity

• Identity field

As noted in the Domain Model section, our DomainModel objects are implemented with a Key parameter that serves to identify them. This Key class contains an array of Objects[], which are the combination of fields that are their primary key in the database. This allows for both simple and compound primary keys.

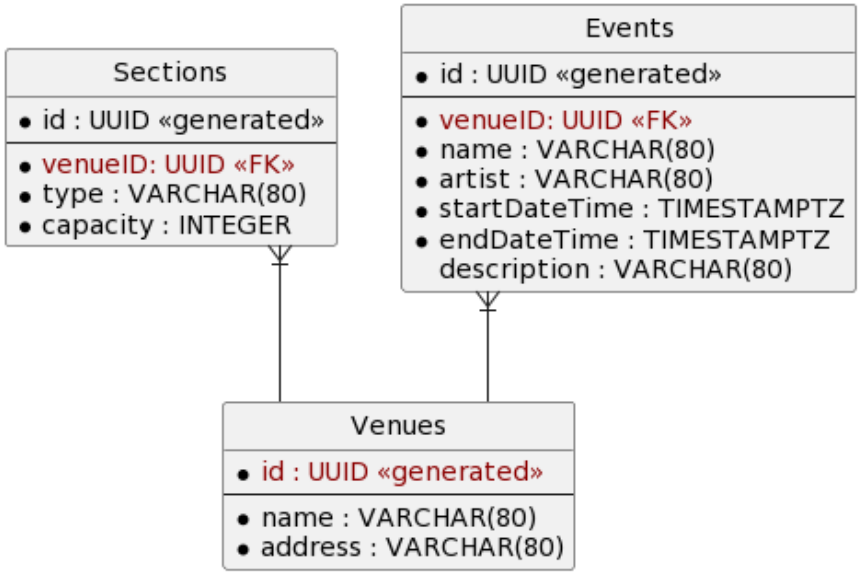
Our Key acts similarly to a layer supertype to the varying primary keys that our objects may have (both simple and compound). It is implemented with the following methods:

- Initialization methods: If given a UUID, register that as its only primary key. If given a set of fields as an Object[], check for any null values and if all values are non-null, register the primary key fields.
- Getters: Get the set of primary key fields (getPks), get an individual UUID primary key if applicable (getId), get a key of a particular index within the field array (getKey)

It thus allows us to generalise the primary-key setting process.

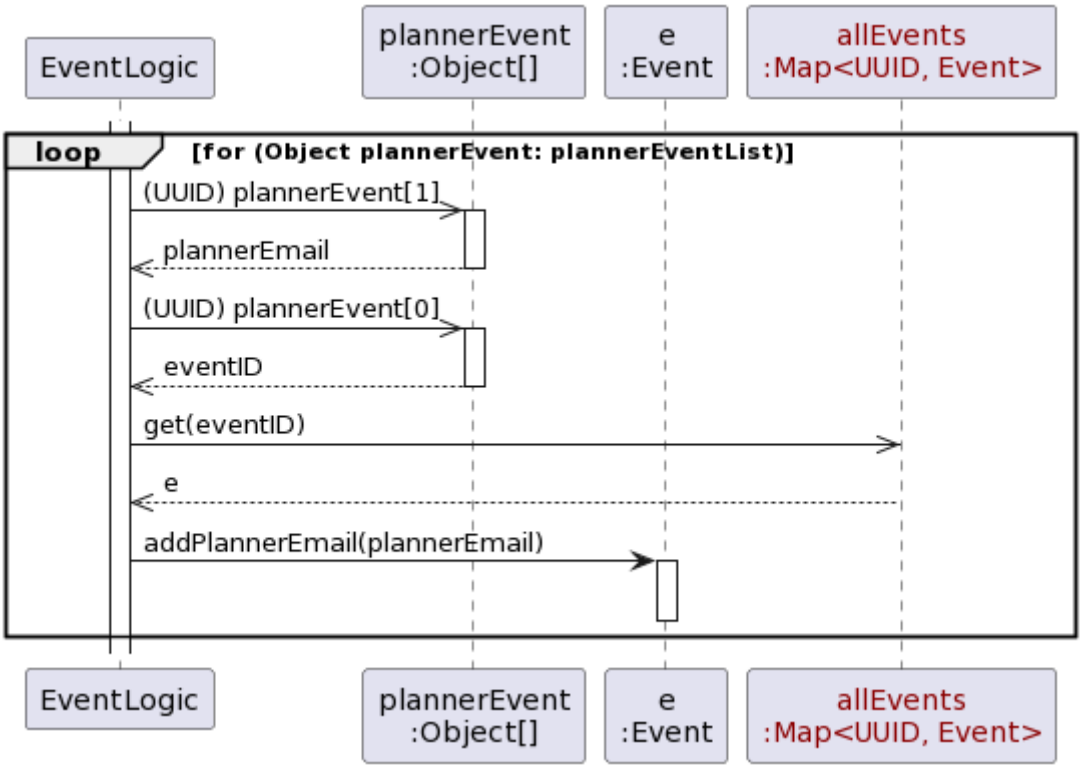
For each of our objects, its Key contains:

- Users
  - Users are naturally differentiated by their login identifier in the business domain (i.e. their identifier is already non-null and unique), so we use their login identifier email (String) as a meaningful primary key for them.
  - Since a user's email is enough to identify a user, we keep it as a simple key.
- Venues
  - In the business domain, venues do not come with a simple primary key. Venues across different locations may have the same name, and vice versa. Thus, we configured the database to automatically generate UUIDs as meaningless primary keys for them.
  - Venues can technically be differentiated by their name and address together (as a combined primary key); however, UUIDs come with the advantage of being simple. This allows other tables to refer to them easily.



A part of our database design, displaying how a Venue's id allows it to relate to other tables in the database.

- Sections
    - Sections are weak entities that belong to Venues and do not naturally come with primary keys. In order to differentiate them, we create a meaningless UUID primary key for each row in the database.
    - We store these keys in our Section objects so that we can pass this information to the frontend and this enable use cases such as Admin Modifies Venues (where they may modify Sections) and Planner Modifies Events (where they may modify the price across sections)
- Events
    - Events do not have primary keys in the business domain. To support frontend UPDATE and DELETE operations, we create meaningless UUID primary keys for them and store them in our objects.
    - Like in Venues, Event keys' simplicity also helps us in creating a class-wide identity map for them in use in View Event use cases. By retrieving all Event rows and storing them in an identity map, the system is then able to easily match EventSection and PlannerEvent data to the Event objects without having to query the database further.



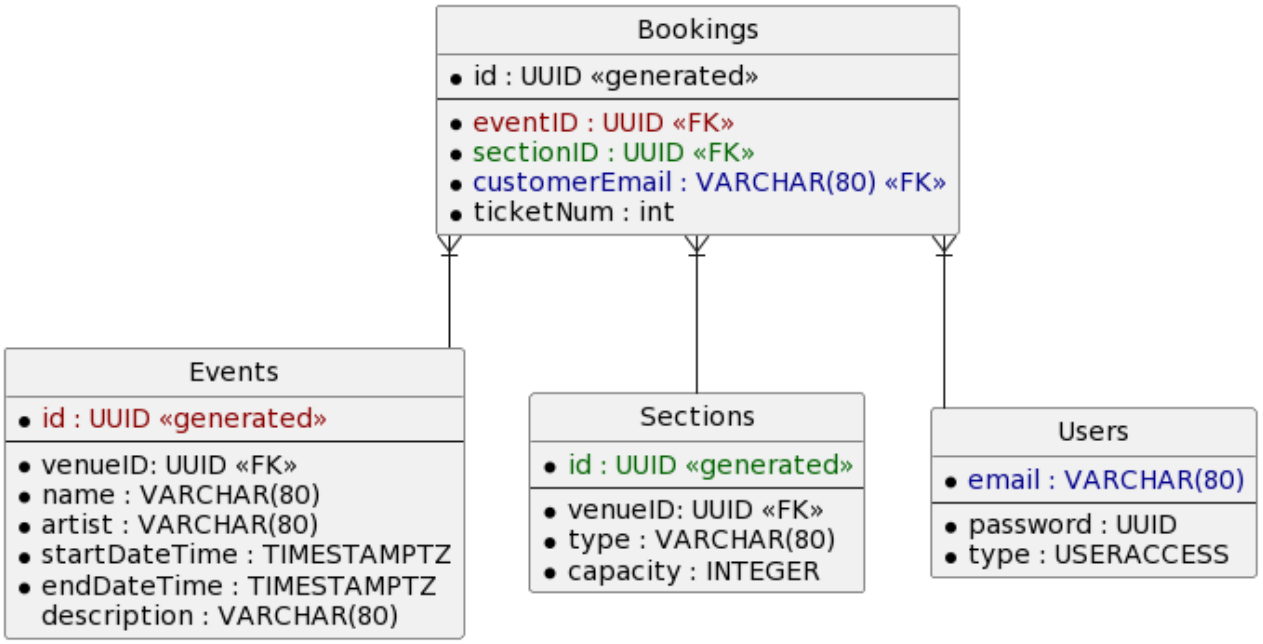
Our use of an identity map for matching PlannerEvent data to Events, during a request for viewing Events. Note that variable names have been changed and irrelevant steps have been omitted for clarity.

- Bookings
    - Bookings do not have primary keys in the business domain, since the same customer may create multiple event bookings in the same section. To support frontend UPDATE and DELETE operations, we create meaningless UUID primary keys for them and store them in our objects.

• Foreign key mapping

A number of our objects use the foreign key mapping:

- Sections
    - In the business domain, each Venue may have multiple Sections, while each Section is only located within one Venue, which makes their relationship one-to-many. Thus, we store the Venue's id within each Section as a foreign key.
- Events
    - A Venue may hold multiple Events, while each Event may only be held in one Venue. This is a one-to-many relationship, which we model by storing the VenueIDs within Events as a foreign key.
- Bookings
    - Similarly, each booking may belong to one Event and one Customer. It thus stores EventID and CustomerEmail as foreign keys.
    - For simplicity, we restricted the Book Ticket use case to one Section per Booking, in order to create a one-to-many relationship between Sections and Bookings. We store SectionIDs in Bookings.

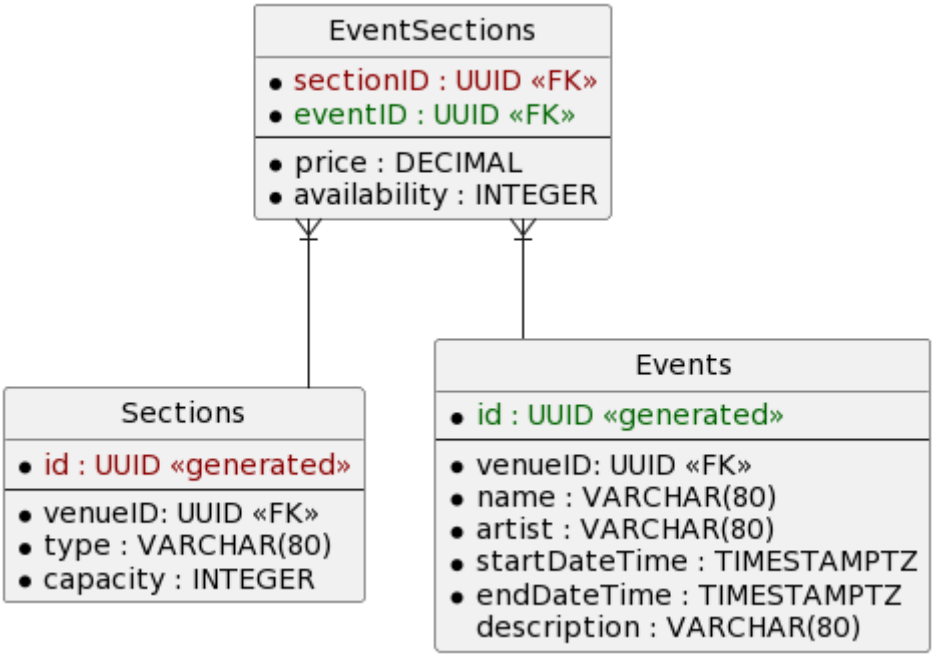


A part of our database design, showing how the Bookings table relates to Events, Sections and Users using their primary keys as foreign keys

• Association table mapping

We created the following tables:

- EventSections
    - Each Event may involve multiple Sections, and each Section may hold multiple events. Thus, their relationship is many-to-many, so we create the table EventSections to model it.
    - As seen in Section's ER diagram table in the Identity Field section, this also allows us to store different pricing and availability for each Section during each Event.

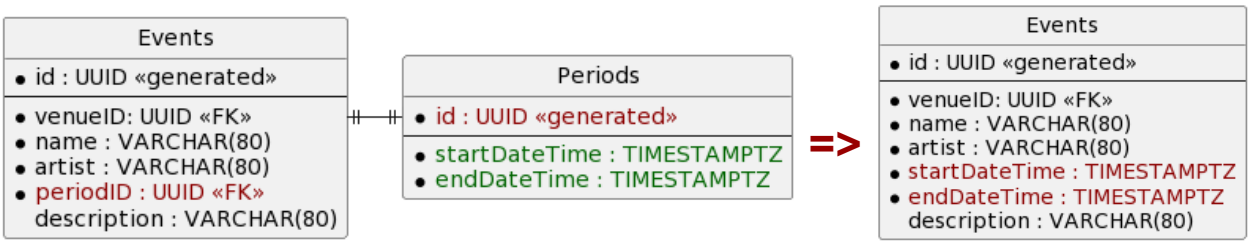


A part of our database design, showing how the many-to-many relationship between Events and Sections, as well as its additional attributes, is modelled as the table EventSections.

- PlannerEvents
    - Each Event may be planned by multiple Planners, and each Planner may plan multiple events. We created the table PlannerEvents to model this.

• Embedded value

In the business domain, each Event runs for a certain Period of time, which is a value object with a start datetime and an end datetime. However, to keep our database clean, we do not create a table for individual time periods; instead, we “flatten” the Period object into a startDateTime and an endDateTime, both of which we embed into our Event table.



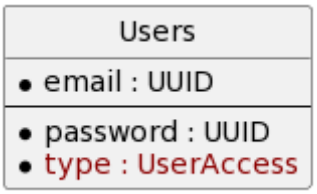
Left: Would-be design of Events and their Periods, if we model them according to the business logic. Right: Our implementation of the database, where we embedded Periods within Events.

This pattern allows us to more easily perform use cases such as Customer View Event Calendar, where we would query Events by their startDateTimes, since we would not have to perform a costly JOIN operation between Events and Periods tables. Our database schema is also less cluttered as a result.

• Single table inheritance

- Admin, Customers and Planners inheriting from Users

- None of them have any fields that are unique to themselves; Customers' relationship with Bookings is one-to-many (and stored by Bookings), while Planners' relationship with Events is many-to-many (and thus stored in an association table PlannerEvents). Hence, in the database each row of these three entities only need to store their email and password.
- This allows us to easily implement the single table inheritance pattern and store them all as Users, which stores email and passwords. An extra field "type" (of custom-created enum type UserAccess) is stored for each row
- These entities' simplicity allows us to bypass most disadvantages of this pattern: No field will be left null (email and passwords are always required), and thus no space is wasted.



A part of our database design, showing how the Single Table Inheritance between User's three subclasses are stored within the same table with an extra "type" field to distinguish the subclasses.

• Authentication and Authorization

Since our application is implemented as separate frontend and backend components, we utilised the Java JWT library to generate JSON Web Tokens upon a frontend client's login, so that the frontend can use them to authenticate itself on our Spring Security-managed backend, and receive authorization to send requests to routes that require certain tier/s of access. Our system ensures that:

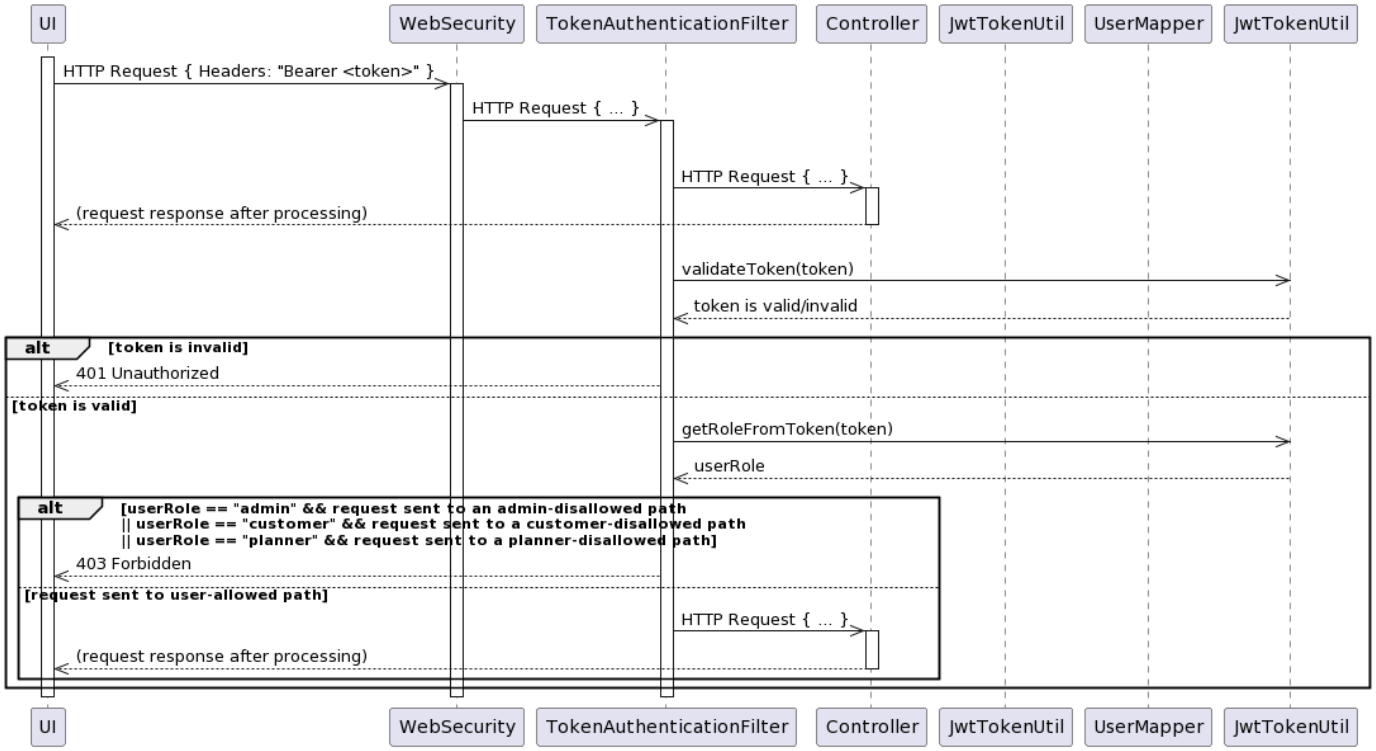
1. Besides the initial login and signup pages, only requests with valid tokens in their header will be accepted. E.g. Non-authenticated Users cannot view Events.
2. Access tokens will be checked against their corresponding users' access level/s, and requests may be rejected if the user does not have the correct authorization. E.g. Customers are not allowed to edit Events.
3. For security purposes, access tokens expire after a short period of time, but the frontend will be able to use a refresh token (received upon login) to first obtain a new access token, then resend a request with the new token and receive data.

In order to enable this, we implement the following:

- Token string, refresh token string, JwtTokenUtil, UserMapper
  - In the User table within our database, we add a field to store access tokens.
  - Our access token and refresh tokens are implemented as Strings for simplicity.
  - Our UserMapper makes requests to store tokens, retrieve and delete refresh tokens by their table id, or delete all tokens for a particular user email.
  - Our JwtTokenUtil class is responsible for creating and validating tokens, as well as parsing given tokens into user email and role data.
- TokenAuthenticationFilter

    - We create the custom TokenAuthenticationFilter class, which rejects and/or accepts requests based on the route and the given token's GrantedAuthority. It uses the aforementioned JwtTokenUtil class to process tokens given to it.
    - We add this Filter within WebSecurityConfig's securityFilterChain() method, so that this filter is run every time the backend processes a request.
- Servlets for login, refresh token and logout
    - Our Login and Logout servlets handle requests from frontend calls. The Login servlet is also tasked with sending information to the frontend to enable refresh token storage on the client side.
- Frontend changes
    - Upon login, refresh and access tokens are generated and sent to the frontend.
    - The frontend stores this information and attaches it in the header of each request made to the backend.

Our process for filtering a request from the client request would thus look like:



Our security system at work to authenticate and authorise a user.

Design rationale for unit of work and lazy load

• Unit of work

We implement a UnitOfWork as a layer of indirection between our controllers and our mappers. It is an important part of our system. Although it is less necessary for VIEW operations, its value comes through for the more complex CREATE-, UPDATE- and DELETE-related use cases:

- CREATE: In our business domain, use cases sometimes "entangle" objects with one another; creation of a Venue also involves creating Sections, and creation of Events involves creating EventSections and PlannerEvents.

- UPDATE: Edits made to a Venue may simultaneously lead to the creation, update and deletion of any number of Sections, and similarly edits made to an Event may affect the associated PlannerEvents table.
- DELETE: Although our implementation of the use cases restrict users to deleting one item at a time, future extensions to the application may require use cases such as batch deletion of bookings etc., in which case the system will have to keep track of all of these deletions at once.

Although these use cases all affect multiple rows at the same time, they are nonetheless individual transactions, and as such must be kept atomic. Our UnitOfWork implementation would help us with managing transactions by storing all changes caused by a transaction, then committing these changes all at once, so the controller does not have to make separate calls to the database during the running of the transaction. It also handles the SQLConnection for this transaction, so it will commit the connection or rollback if any errors occur during the process, so as to ensure partial transactions are not being performed.

Moreover, as we expand the application for use by a large number of users, concurrency control will become increasingly important to ensure that our transactions are both consistent and isolated from each other, especially since we have many use cases that may lead to one user tampering with data that another user may be using:

- Multiple planners editing event: Since multiple planners may edit the event at the same time, making many small, sparse requests to the database will more likely lead to requests interleaving with each other, which can cause users' updates to be lost and/or cause the row data to become inconsistent.
- Planner/s editing event while customers book tickets: It is possible for an event's data to be changed during a customer's booking session. Thus, it is essential for the system to track if the database information has changed prior to creating a new booking for the customer.
- (Various other use cases such as admin editing venues can also lead to similar issues as in the two examples above.)

The UnitOfWork is able to protect against these to a certain extent; it can be implemented to perform concurrency checks, and it can also keep track of previously read objects and compare them against the database, so it can rollback/redo any transactions that may be affected by changed data.

Furthermore, the UnitOfWork acts to increase the controller logics' cohesion. By taking over the responsibility of calling the actual mappers and handling database connections, the controllers only need to handle the actual logic.

Lazy load

Lazy load

For many of our use cases, we do not need all of a business object's data to be loaded at once:

- Customer views booking: To prevent errors from event updates, we do not store any information on the event in the booking object, which means that for the use case to be practical for customers we need to make an extra call to the server to retrieve the booked event's name (and arguably start/end datetimes and address). However, other information

such as per-section prices/availabilities (EventSections) and responsible planners (PlannerEvents) aren't relevant to the customer at all at this point.
- Admin views users: When viewing the list of users, the administrator does not need to have customers' booking information/planners' event information available all at once; they should be able to choose to view particular users' data only if needed.
- (Various other use cases follow the same logic as the above examples.)

Thus instead of loading all data at once, which will slow down the application exponentially as the amount of data and number of users grow, we implement the lazy load pattern in our DomainObject superclass, so all of our objects can benefit from this pattern. This allows us to load only the necessary parts of the data, and thus speed up our operations significantly as our amount of data grows.

Although in certain scenarios lazy loading may lead to ripple loading and thus slow down transaction speed, our business domain is simple enough that for each transaction, we already know what information is required from the get-go, so there won't be any extraneous loads:

- CREATE: We occasionally need information from newly-created objects to support creating other associated objects (e.g. venueID for sections). However, through the "`RETURNING`" SQL keyword we automatically receive any necessary information, which is enough to support our use cases.
- READ: None of our use cases have any logic that changes what needs to be returned, based on initially-retrieved data. One may argue for cases such as filtering data (e.g. viewing events within a 6-month period), but the "`WHICH`" SQL keyword is sufficient for our logic.
- DELETE: Although some of our deletion use cases affect objects in other tables (e.g. deleting events leading to deleted bookings), by setting our foreign key constraints to "`CASCADE`" on deletes, we ensure that deletion of associated objects is handled automatically by the database, so we do not need additional information from deletion.
- UPDATE: Similar to the other operations, our business logic is simple enough that this does not require additional data.

As such, the lazy loading pattern will be perfect for our needs, and its downsides will not cause issues for our project.

Value holder

Value holder's biggest trait is that unlike virtual proxies and ghosts, it is not of a particular type (e.g. Proxy-Section), but is instead a more generic wrapper.

In the context of our system, other methods have the following disadvantages:

- Lazy initialization: Since this requires direct access between the object and the database, it is incompatible with our Mappers
- Virtual Proxy: This is less generalisable than the Value Holder and allows less room for expansion
- Ghost: As described above, we often do not need the entire object loaded at once.

Therefore, the value holder is the most suitable for our application.