1. Descripción general

La aplicación UAT-Tool es una herramienta para la gestión y ejecución de pruebas (User Acceptance Testing) del sistema U-space de ENAIRE. Permite organizar campañas de pruebas, gestionar casos de uso, ejecutar tests y registrar bugs encontrados.

1. Modelos de datos

La representación gráfica de la estructura de tablas de la base de datos del programa se puede consultar en el documento html situado en esta misma carpeta.

Las tablas de la base de datos pueden organizarse según la sección del programa con la que guardan relación. Estas secciones son:

* Assets: activos o elementos que son empleados en la realización de las pruebas. También hace las veces de inventariado de dichos elementos registrados en el sistema U-space de Enaire:
  + Emails *(emails)*
  + Operadores USSP *(operators)*
  + Drones *(drones)*
  + Organizaciones U-hub *(uhub\_orgs)*
  + Usuarios U-hub *(uhub\_users)*
  + Zonas UAS *(uas\_zones)*
  + U-spaces *(uspaces)*
* Requirements *(requirements)*: requisitos de usuario que afectan a los diferentes elementos del sistema U-space de Enaire.
* Test Management: agrupa las tablas que constituyen los elementos que definen los casos de uso y pasos a seguir por el ejecutor de las pruebas sobre el sistema:
  + Steps *(steps)*: pasos a seguir por parte del usuario a la hora de llevar a cabo una prueba sobre el sistema.
  + Cases *(cases)*: casos de uso de pruebas a realizar sobre el sistema. Agrupan una serie de pasos (steps).
  + Blocks *(blocks)*: bloques de casos de uso. Permite agrupar casos de uso (cases) para su incorporación en bloque a las campañas de pruebas.
* Campaigns (campaigns): Campañas de prueba. Agrupa una serie de bloques (blocks) de casos de uso y crea un borrador que luego puede ser ejecutado (campaign run)
* Executions: agrupa las tablas relacionadas con las ejecuciones de campañas de prueba y que almacenan los resultados de estas para su posterior post procesado:
  + Campaign Runs *(campaign\_runs)*: representa la ejecución de una campaña de pruebas. Permite conocer a qué hora se comenzó y terminó y quién la ha modificado por última vez. Su relación con los case y step runs permite saber el porcentaje de éxito de la campaña.
  + Case Run *(case\_runs)*: representa el estado de ejecución de un caso de prueba dentro de una campaña.
  + Step Run *(step\_runs)*: representa la ejecución de un paso dentro de un caso de uso en una campaña. Permite indicar si el estado está aprobado o no y adjuntar notas y archivos mostrando los posibles errores en la ejecución.
* Bugs *(bugs)*: almacena los errores encontrados en la plataforma. Su incorporación a la tabla puede ser realizada manualmente mediante el usuario o generarse automáticamente a partir de los resultados de una campaign run. Adicionalmente a la información básica del bug, existe una tabla dependiente de la tabla *bugs*, llamada *bug\_history,* que almacena los cambios de estado de un bug entre los posibles (abierto, resuelto, cerrado, etc).

A parte de las tablas mencionadas anteriormente, existen unas que, si bien no son modificables por el usuario vía interfaz, sirven como listas de valores por defecto para algunos de los campos de otras tablas, o son tablas auxiliares para el funcionamiento del programa. Estas tablas, llamadas *auxiliares* son:

* Environment *(environments)*: representa un entorno de instalación del sistema U-space de Enaire. Esta tabla sirve para particularizar las pruebas y los bugs por entorno. La creación de un nuevo entorno se hará mediante instrucción directa a la base de datos, y el usuario al iniciar la aplicación podrá elegir sobre qué entorno quiere ejecutar las acciones.
* Systems *(systems)* y Sections *(sections)*: listas de sistemas y secciones predefinidos según las pruebas que se han realizado anteriormente sobre el sistema U-space. Entre los sistemas se encuentran el USSP, AUDI, ExServices, etc. Entre las secciones se encuentran aquellas a las que afectan los requisitos (NID, FP Auth, etc). Estas listas no son modificables por el usuario y únicamente sirven para alimentar los diferentes formularios de elementos que dependen de estas tablas.
* Reasons *(reasons)*: lista de razones de creación de una zona UAS según el estándar ED-318. Esta lista alimenta el formulario de creación de zonas UAS.
* File *(files)*: tabla que almacena los diferentes archivos vinculados a U-spaces o bugs que existen en el sistema. Las demás tablas se vinculan con esta mediante el id del archivo en la tabla *fiiles* en lugar de almacenar el archivo en su propia tabla.

Cada tabla tiene unas columnas determinadas según lo necesario para su uso, pero la mayor parte de ellas comparten una serie de columnas comunes. Estas columnas comunes son:

* id: número entero que identifica el elemento en cada tabla, clave primaria y autoincrementado con cada nuevo registro.
* created\_at: fecha en la que se creó el registro, siendo esta por defecto la del servidor de base de datos en el momento en el que se realiza la creación *(server\_default=func.now()).*
* updated\_at: fecha en la que se realiza una actualización sobre el registro, con dos fechas, una primera por defecto cuando se genera la creación en Python en el equipo *(server\_default=func.now())*, igual que la de creación y otra de actualización siendo por defecto la del equipo (ejecutada en Python cuando se realiza la acción) *(onupdate=func.now())*.
* modified\_by: usuario que ha realizado el registro o modificación del mismo en último lugar.
* environment\_id: entorno en el cual está registrado o al cual hace referencia el elemento. Hace referencia al id del entorno creado en la tabla *environments*.

Para incluir de manera heredada estos elementos en todas las tablas y evitar repeticiones, se han creado dos *Mixin* en SQLAlchemy (*AuditMixin* y *EnvironmentMixin*). Todos los modelos están basados en estos mixins y en una Base declarativa de SQLAlchemy excepto:

* CampaignRun: hereda únicamente de Base y de *EnvironmentMixin*, ya que el modelo tiene su propio campo de *modified\_by* y sus campos de fechas de comienzo de campaña, actualización y finalización.
* CaseRun y StepRun: únicamente heredan de Base, ya que dependen directamente de CampaignRun y heredan los campos de auditoría y entorno de la campaña a la que pertenecen.
* Step: solo hereda de Base, ya que depende directamente de Case y hereda los campos de auditoría y entorno de este.
* BugHistory: solo hereda de Base, ya que depende directamente de Bug y hereda los campos de auditoría y entorno de este.

En cuanto a las relaciones entre tablas, existen de diferentes tipos:

* Relaciones de uno a uno: por ejemplo, un operador deberá estar vinculado obligatoria únicamente a un email en cada entorno. Estas relaciones se representan en los modelos SQLAlchemy mediante la columna email\_id, por ejemplo, que contiene los ids de la tabla padre con la que se relaciona.
* Relaciones de muchos a muchos: por ejemplo, una zona UAS podrá tener varias organizaciones de U-hub vinculadas y, a su vez, una organización podrá estar organizada a múltiples zonas. Estas relaciones se representan en los modelos SQLAlchemy mediante tablas intermedias (presentes en el archivo *associations.py* en el proyecto). Estas tablas almacenan en cada fila los ids de cada elemento por columna.

Además, en algunas tablas se incorporan las llamadas restricciones. Restricciones como *UniqueConstraint* establecen la condición mencionada anteriormente de que no existan más de un registro con el mismo valor para un mismo *environment\_id*. Además, existen otras restricciones como *CheckConstraint* en modelos como el de zona UAS, que fijan las condiciones de *nullable=False* para campos como el radio del círculo cuando la zona es de tipo CIRCLE y no en todos los casos.

Ciertos modelos pueden incluir métodos específicos para comportamiento intrínseco del registro. Por ejemplo, el modelo de bug puede contener un método para cambiar de status directamente en lugar de necesitar ejecutar el método de actualizar registro con únicamente un cambio de status. Todos los métodos implementados directamente en el modelo deben depender únicamente de validaciones sobre datos internos o cálculos derivados de atributos propios. En el momento en el que es necesario realizar operaciones que involucran múltiples entidades será necesario hacerlo mediante los servicios.

1. Setup de base de datos

La base de datos está configurada sobre SQLite, para poder disponer de ella sin necesidad de instalarla en un servidor (el archivo de base de datos estará alojado en la propia carpeta de datos del programa, el cual será ejecutado por diferentes usuarios desde una carpeta del OneDrive de ENAIRE.

Para su manejo con Python y evitar el uso directo de líneas de código en SQL, el programa hace uso de la librería SQLAlchemy, la cual permite, mediante el denominado ORM (Object Relational Mapping), usar clases Python para el manejo de la base de datos.

La configuración de la base de datos comienza con la creación de un motor (*engine*), mediante el método *create\_engine* y la url de la base de datos. El motor establece la conexión con la base de datos, maneja los recursos de dicha conexión, traduce las operaciones Python-SQL y gestiona la conexión de múltiples usuarios.

Una vez establecido el motor, se crea un objeto *Session* mediante el método *scoped\_session(sessionmaker(bind=engine))*. La sesión representa la “mesa de trabajo”, donde se acumulan los cambios (como un carro de la compra) antes de guardarlo definitivamente mediante *commit()*. La sesión se crea como *scoped* para que las diferentes sesiones instanciadas no se mezclen si el programa usa hilos de ejecución.

Finalmente, existe el elemento llamado Base, creado mediante el método *declarative\_base()*, que constituye el molde del que heredan los diferentes modelos explicando en el punto anterior.

MIGRACIONES/ALEMBIC PENDIENTE DE IMPLEMENTAR Y EXPLICAR

La inicialización de la base de datos tiene lugar dentro del flujo de inicio del programa. Para ello, existe el método *init\_db(drop\_existing:bool, engine)*, el cual emplea los elementos de configuración de la base de datos presentados anteriormente para crear las tablas (*Base.metadata.create\_all(db\_engine))* si no existen (*drop\_existing* en True provoca el borrado de las tablas antes de su nueva creación).

Dentro de este método se realiza una llamada al método *load\_initial\_data(session)*, que permite cargar (si no existen ya) los valores por defecto para los Systems, Sections y Reasons.

1. Patrón repositorio

El programa sigue un patrón de comunicación con la base de datos mediante el uso de repositorios. Los repositorios consisten en diferentes clases que contienen métodos para gestionar los elementos de la base de datos de manera que no exista código de SQLAlchemy esparcido por el proyecto, lo cual permite una mejor separación de responsabilidades y un código más limpio. Las consultas realizadas por el programa a través de los servicios son procesadas por el repositorio, en lugar de emplear los propios servicios para lanzar consultas directas a la base de datos.

De este modo, se ha creado un repositorio base que sirve como clase padre de los repositorios más concretos según las diferentes tablas de la base de datos. Este repositorio base desarrolla las operaciones CRUD básicas que son comunes a cualquier tabla e incorpora manejo de errores y algunas funcionalidades más allá del CRUD (Create, Remove, Update, Delete) básico. Algunas (no se incluyen todas) de estas funciones básicas son:

* *get\_by\_id*: obtiene un registro dado el identificador dentro de la tabla.
* *get\_all*: obtiene todos los registros de una tabla (con opciones para seleccionar un número máximo de registros o un offset).
* *filter\_by:* filtra registros según los criterios proporcionados (campo=valor).
* *create:* crea un nuevo registro en la base de datos dados los atributos del nuevo registro.
* *update:* actualiza un registro existente dada la instancia a actualizar y los datos del nuevo registro.
* *delete:* borra un registro existente dado su identificador.

Existen, además, dos clases de repositorio que heredan de la clase base: *AuditMixinRepository, EnvironmentMixinRepository y AuditEnvironmentMixinRepository*. Cada uno de estos nuevos repositorios implementan funciones para crear y actualizar registros comprobando que los datos a introducir contienen los valores necesarios según su modelo de datos herede de *AuditMixin*, *EnvironmentMixin* o ambos.

Finalmente, cada tabla contiene su propio repositorio, el cual hereda del repositorio base y los mixins según corresponda. Adicionalmente a los métodos implementados por las clases padre, cada repositorio contiene funciones específicas según las necesidades de cada tabla. Por ejemplo, el repositorio de Campaigns permite la creación de una campaña y vincula los pasos, casos y bloques de pruebas con la misma.

1. Patrón U*nit of Work (UoW)*

Adicionalmente al uso de repositorios, el programa implementa también la llamada *unit of work*, que permite simplificar las operaciones que los servicios hacen en la base de datos a través de los repositorios. Este patrón establece las llamadas transacciones atómicas, que permiten ejecutar todo o no ejecutar nada de lo cambiado si existe algún error, en lugar de realizar todas las acciones individualmente con sesiones diferentes. Todos los repositorios comparten la misma sesión, y la unidad de trabajo maneja todas las operaciones de manera global mediante *commit, rollback o close*.

Por ejemplo, la unidad de trabajo permite ejecutar la creación de dos elementos en la misma sesión (ex: *uow.email\_repo.create(email)* y *uow.campaign.repo.create(campaign*)) y realizar un *commit* de todo automáticamente o, si algo falla, hacer *rollback* de todo de la misma manera.

Además de la propia clase UnitOfWork, existe el *@contextmanager* que actúa como un asistente que genera una sesión, crea el uow y lo proporciona a la función que lo necesite (en su contexto). Si todo va bien realiza el *commit* o si falla lo cancela todo *rollback*. Gracias al context manager no es necesario instanciar la uow, sino que bajo el contexto de la función se trabaja con el uow.

Los servicios usarán el uow a través del context manager en lugar de comunicarse directamente con los repositorios.

1. Capa de servicios

Los servicios se encargan de la lógica de negocio de la aplicación. Actúan como nexo entre las órdenes del usuario a través de la vista y el controlador y la base de datos (con la uow como punto de entrada). Además, previo a ejecutar las acciones sobre la base de datos, se encargan también de la verificación de los datos y la aplicación de las reglas sobre estos *(constraints)*.

Cada servicio se ajusta a las necesidades propias de la sección o repositorio al que sirven en su empleo. A través de estos se pueden obtener registros de la base de datos (todos o selección por id) bajo el contexto de la unidad de trabajo, por ejemplo:

*def get\_all\_bugs(self) -> list[Bug]:*

*"""Obtiene todos los bugs como objetos SQLAlchemy (para lógica de negocio)."""*

*with self.app\_context.get\_unit\_of\_work\_context() as uow:*

*return uow.bug\_repo.get\_all()*

Además, también se pueden crear, actualizar o borrar registros.

Los servicios ofrecen dos formas de trabajar con los datos según el contexto:

* Objetos SQLAlchemy*:* Para operaciones complejas que requieren comportamiento de negocio
* DTOs*:* Para transferencia segura de datos hacia la interfaz u otros servicios"

Los objetos de SQLAlchemy permiten usar métodos de negocio implementados directamente en la base de datos. Por ejemplo, si se ha implementado un método en la clase Bug que permite cerrarlo (cambiar su estado a CLOSED), trabajar con el objeto SQLAlchemy permite realizar una llamada a dicho método directamente. Homólogamente, también se puede acceder directamente a propiedades del objeto, por ejemplo, con bug.campaign\_id.

Los DTOs (Data Transfer Objects) sirven para una transmisión de datos más directa a la interfaz, por ejemplo, de manera que la interfaz no tenga que realizar operaciones sobre la base de datos directamente. Los DTOs también son más útiles que los objetos SQLAlchemy para separar responsabilidades entre servicios diferentes y aun así poder transmitir datos. Los DTOs presentan ventajas sobre los diccionarios ya que los datos son tipados y permiten una auto documentación (siempre se sabe qué datos son necesarios al crear el objeto). Además, se pueden incluir métodos de autovalidación dentro de la propia clase.

Las validaciones de los datos a introducir en la base de datos a través de los servicios se pueden realizar mediante métodos en los propios servicios o ser directamente validada en los DTOs.

La validación en DTOs se realiza para verificar aspectos como:

* Longitud de campos.
* Formatos (email, fechas).
* Enumerados predefinidos (el valor debe estar entre unos valores concretos).
* Rangos numéricos simples (el valor debe ser mayor que cero).

La validación en servicios se realiza para situaciones como:

* Reglas de negocio complejas.
* Verificaciones en base de datos (un bug es muy similar a otro).
* Permisos de usuario (un usuario no puede modificar determinados datos).
* Límites empresariales (no se pueden almacenar mas de x registros en un día.

1. App Context

El llamado Application Context actúa como un cerebro central que contiene las dependencias y contexto de la aplicación. Gestiona la lógica de negocio, datos y configuración, pero no tiene conocimiento sobre las interfaces de usuario, simplemente se enfoca en la lógica interna.

Por un lado, inicializa dichas dependencias, es decir:

* La base de datos: mediante la llamada del método init\_db explicado anteriormente.
* Los servicios de cada sección de la aplicación: instancia dichos servicios y los registra como inicializados.

También incorpora un panel de configuración con variables sobre la sesión de base de datos, los servicios inicializados y un flag de inicialización del propio contexto.

Además, también presenta un “ciclo de vida”, es decir, se inicializa mediante *initialize(self)*, contiene funciones útiles para su operación normal, como por ejemplo *get\_uow* o *get\_service* y proporciona un método para su cerrado de manera limpia (*shutdown)*, el cual cierra el contexto y libera recursos de servicios cerrándolos también.

Por último, existe un método *@classmethod* destinado a que exista un único contexto de aplicación (patrón singleton). Siempre que otra función llame a *get\_instance* se devolverá la misma instancia de ApplicationContext.

1. Inicialización del programa

En esta sección se explica el flujo de iniciación del programa, aunque no se explica, de momento, la inicialización desde un punto de vista de interfaz de usuario, simplemente a nivel de lógica interna. El inicio del programa se realiza a través del archivo *main.py*, el cual define como elemento principal para este punto de entrada el *orquestador*.

El orquestador coordina el flujo completo de la aplicación, conectando el cerebro lógico *(ApplicationContext)* con la interfaz de usuario. Gestiona el ciclo de vida, inicialización ordenada y la comunicación entre capas. Esto garantiza la separación de responsabilidades y que cada elemento no tenga por qué conocer los elementos de otras capas ajenas del programa.

Por tanto, el punto de entrada a la aplicación es la instanciación del orquestador, el cual como parámetros incorpora la propia aplicación y el contexto de esta (además de los controladores y vistas, que ya se explicarán al hablar de PySide6 y la experiencia de usuario).

Inicialmente se ejecuta *orchestrator.initialize\_application*, que realiza las siguientes acciones:

* Configura y crea la base de la aplicación (en concreto la aplicación QT, la interfaz gráfica)
* Inicializa el contexto de la aplicación (*app\_context.initialize*). Este proceso viene explicado en la sección anterior más en detalle.
* Configura el manejo de excepciones global. Si alguna excepción del programa no está capturada, registra el error en el log y evita que la aplicación se rompa inmediatamente.

El orquestador también incorpora un método de arranque, que es ejecutado tras su instanciación, y un método de cierre seguro *(\_safe\_shutdown),* que se ejecuta cuando el usuario cierra la aplicación (cerrando la ventana principal mediante el método *shutdown* propio, la base de la aplicación mediante *app.quit* y el contexto mediante su propio método *shutdown*).

1. Modelo Vista Controlador – Interfaces UI con PySide6

Además de la inicialización del contexto de la aplicación, que a su vez configura los servicios, el script de arranque inicializa también la pantalla principal del programa.

La librería responsable de las interfaces de usuario es PySide6, un conjunto de paquetes que proporciona el marco para la interacción con el usuario y la presentación de datos mediante elementos como tablas, listas, formularios, entre otros. Internamente, PySide6 opera bajo un modelo de vista-modelo (no confundir con el patrón MVC de la arquitectura general del sistema), en el cual cada vista (tabla, lista o pantalla de datos) se alimenta de un modelo de datos configurado en el *backend*. Cualquier modificación en dicho modelo se refleja automáticamente en la vista, permitiendo al usuario manipular y consultar los datos de manera fluida e inmediata.

Por otro lado, la gestión de la interfaz de usuario a nivel arquitectónico sigue el esquema de MVC (Modelo-Vista-Controlador). En este contexto:

* La Vista, que hereda de una pantalla diseñada gráficamente en QTDesigner, se limita exclusivamente a tareas relacionadas con la interacción directa del usuario —como la captura de clicks y otras acciones en la interfaz—. No contiene lógica de negocio ni conoce más allá de la presentación de datos y la emisión de señales ante eventos de UI.
* Los Controladores actúan como receptores y procesadores de dichas señales. Cada vista cuenta con un controlador asociado, cuya responsabilidad es escuchar los eventos de la interfaz, coordinar con los servicios correspondientes y actualizar la vista cuando sea necesario.
* Los Servicios, ubicados en la capa de aplicación, implementan las reglas de negocio y orquestan el acceso a los datos, comunicándose con la base de datos a través de los repositorios. De esta forma, se garantiza una separación clara entre la lógica de presentación, la de aplicación y la de dominio.

Esta estructura permite una clara separación de responsabilidades: las vistas se ocupan únicamente de la UI, los controladores manejan el flujo de interacción, y los servicios encapsulan la lógica de negocio y el acceso a datos, facilitando el mantenimiento, la escalabilidad y la testeabilidad del sistema.

1. Main Window – Vista principal de la aplicación

El punto de entrada de la aplicación a nivel de usuario es la página principal. Esta página consiste en 6 secciones, cada una de ellas dedicada a la gestión de un elemento de la aplicación (bugs, campañas, gestión de tests, requisitos, activos y resultados (estadísticas)).

Siguiendo el modelo explicado anteriormente, existe una vista para la página principal *MainWindow* y un controlador principal asociado a esta *MainController*. Además, cada sección de la página tiene un servicio dedicado, así como un modelo de tabla para mostrar los datos asociados (por ejemplo, *BugService, BugTableModel).*

El controlador principal dispone de tres señales:

* *tab\_changed:* disparado cuando se cambia de pestaña, con un string indicando el nombre de la nueva pestaña.
* *application\_ready:* disparado cuando se ha producido la inicialización exitosa de la aplicación.
* *application\_error:* disparado si ocurre un error en la aplicación, con un string indicando el tipo de error, lo que permite mostrar un mensaje en pantalla.

Al inicializar, el controlador define como parámetros:

* *app\_context:* contexto de la aplicación.
* *\_current\_tab:* variable interna que indica la pestaña abierta en ese momento.
* *\_tab\_controllers:* variable interna que almacena los controladores de pestaña activados.
* *\_is\_initialized*: booleano interno que indica si el controlador principal ha sido inicializado o no.

La iniciación de los controladores de tab sigue el siguiente flujo:

* El controlador principal instancia y registra en *\_tab\_controllers* cada controlador y conecta las señales de error con el método de log correspondiente (*\_on\_tab\_error*).
* La variable *\_is\_initialized* cambia a *True* y se emite la señal de *application\_ready*.
* Por último, se activa la pestaña inicial por defecto (bugs).

Los controladores específicos para las pestañas heredan sus características de una clase Base (*BaseTabController).* Esta clase incorpora una serie de señales comunes para todas las pestañas:

* *data\_loaded:* señal que indica que se han cargado una serie de datos de la BBDD. Proporciona como parámetro una lista de items cargados para su muestra en la vista (tabla).
* *item\_created:* emite un elemento de nueva creación cuando se crea.
* *item\_updated:* emite un elemento que ha sido actualizado.
* *item\_deleted:* emite el identificador de un elemento eliminado-
* *error\_ocurred*: informa de un error durante alguna de las operaciones coordinadas por el controlador. Emite una string con la información del error. Se conecta directamente con la ejecución de \_on\_tab\_error en el controlador principal.
* *loading\_state\_changed:* indica del cambio de estado en la carga desde base de datos por parte del servicio.

Además, la clase base también emplea variables propias como el nombre de la pestaña, una variable interna indicando si se encuentra activa, y otra con una lista de los datos actuales de la pestaña. Estas variables alimentan los métodos comunes de la clase. Esto métodos son los siguientes:

* *activate*: se ejecuta al activarse la pestaña. Cambia el booleano de *\_is\_active* y lanza un mensaje de información de logging para indicar la activación de la pestaña.
* *deactivate:* comportamiento contrario al método anterior.
* *refresh\_data:* recarga los datos de la pestaña (ejecuta el método de *load\_data ()*)
* *load\_data:* carga los datos de la tabla. El proceso a seguir es el siguiente
  + Se emite la señal de *loading\_state\_changed()*
  + Se ejecuta el método *get\_all\_items(),* que se implementa en cada controlador particular pero que obtiene los ítems de la tabla mediante el servicio correspondiente.
  + Actualiza la variable que almacena los datos actuales de la pestaña y emite la señal de *data\_loaded* con los elementos cargados.
* Existen además tres métodos específicos para las operaciones CRUD, cada uno de ellos dedicados a crear, actualizar o borrar un elemento de la tabla de la pestaña. El proceso común es el siguiente:
  + Se ejecuta el método particular para la operación, que es implementado por cada controlador.
  + Se emite la señal de elemento creado, actualizado o eliminado.
  + Se reflejan los cambios en la tabla ejecutando el método *load\_data.*
* Por último, cada controlador tiene un método de shutdown que limpia la variable de \_current\_data.

Un ejemplo de controlador particular que hereda de la clase base es *BugTabController*. El controlador particular, además de heredar el contexto marcado por la clase madre, implementa instancias adicionales, propias del manejo de bugs:

* *BugService:* la clase encargada de realizar las operaciones de intercambio de datos y lógica de negocio con la base de datos.
* *BugTableModel:* la clase que gestiona el modelo de datos de la tabla que se muestra en la pestaña, de acuerdo con la relación de Model View Model que se explicó anteriormente para PySide6.
* *BugProxyModel:* la clase que gestiona el modelo proxy que se asocia al modelo de tabla y permite que se realicen acciones de filtrado y búsqueda sobre los datos del modelo de la tabla de bugs.

Además, el controlador de la pestaña de bugs implementa su propio método para cargar datos *load\_data* que sobrescribe al método de la clase base. Este método sigue el siguiente flujo:

* Se emite la señal de *loading\_state\_changed()*
* Se obtienen los bugs desde el servicio empleando el método get\_all\_bugs\_dto)=