EMBArk - Software architecture

Summary

EMBArk is a spin-off product of EMBA (https://github.com/e-m-b-a/emba), a security analyzer for firmware of embedded devices by Siemens Energy. EMBArk extends EMBA by providing a web-application, in other words, a user interface to use EMBA. Our task is to extend EMBArk with orchestration: A user can set up several worker nodes and configure EMBArk to use those nodes to execute firmware scans using EMBA. Prior to this project, EMBArk only supported analysing one firmware at a time.

As this project extends an existing software project (EMBArk), we only cover components in the following graphs and descriptions which are either expected to be modified during our project, or, are needed for consistency.

Technology Stack

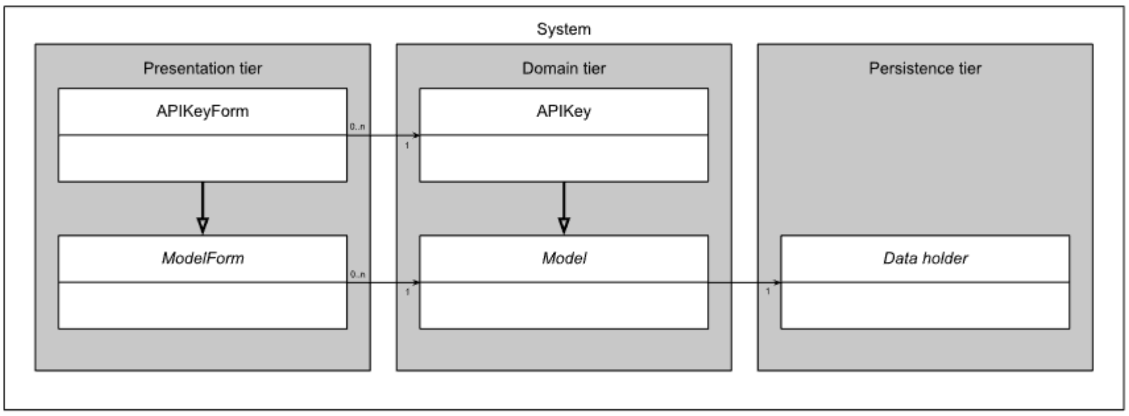
EMBArk is a Django web-application, thus written in Python. It is served using the Apache Webserver (in a production environment), or Daphne Django (in a development environment). Besides Django, EMBArk uses both Bootstrap and jQuery in the frontend.

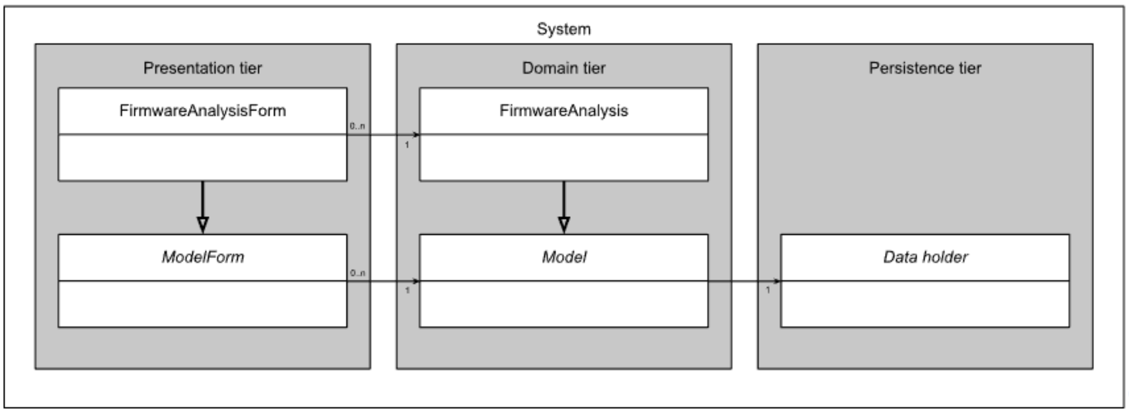
Some API endpoints have to be implemented. These endpoints are tested using a REST client like Postman, and documented using the OpenAPI Specification.

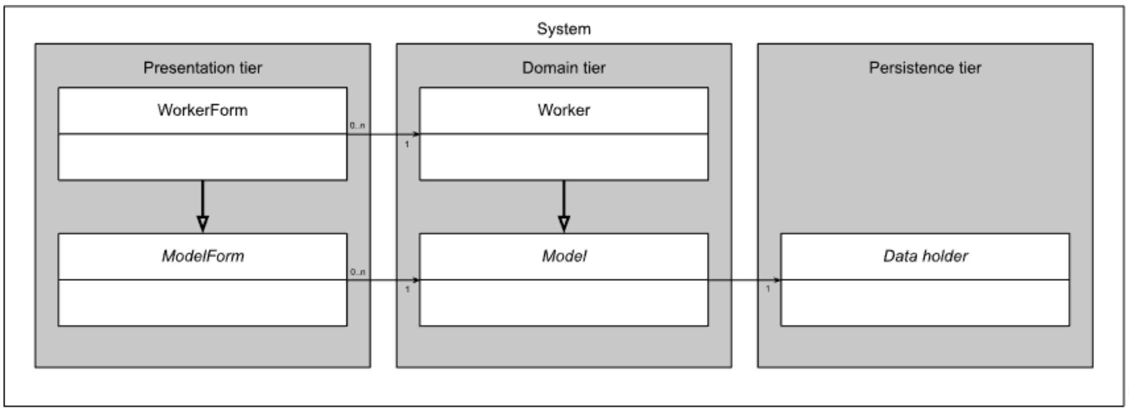
In the backend, the web-application uses a Redis and MySQL server to store data, both managed via Docker. EMBArk manages firmware analysis scans on worker nodes. Each worker runs either Ubuntu 22.04, Ubuntu 24.04 or Kali Linux. Connection to these nodes is done through *ssh* and *sshfs*. During development phase, such workers are provided using virtual machines (e.g. VirtualBox, QEMU).

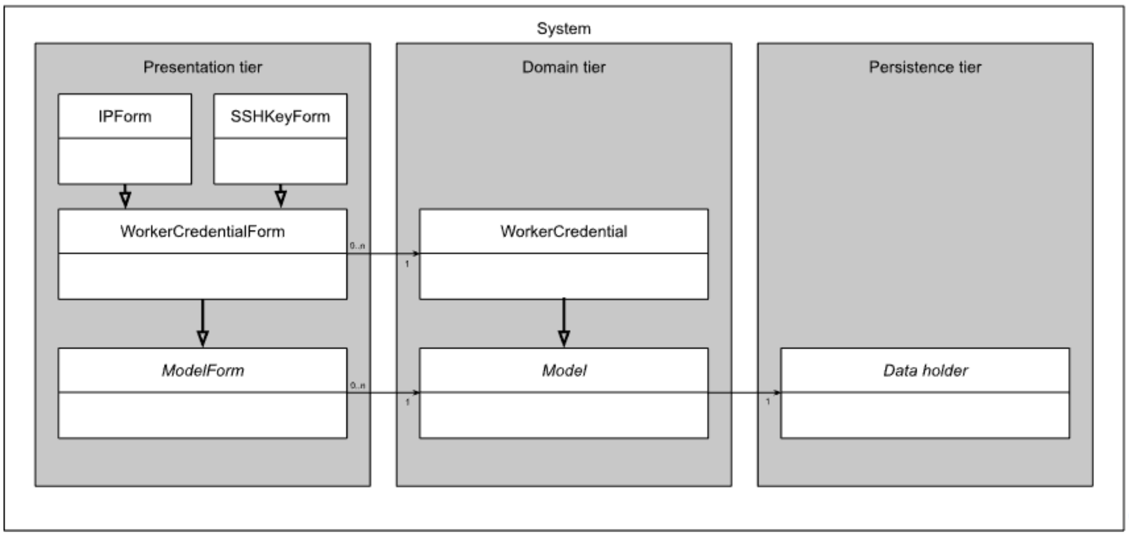
Collaboration and source code versioning is performed using *git*.

Runtime architecture

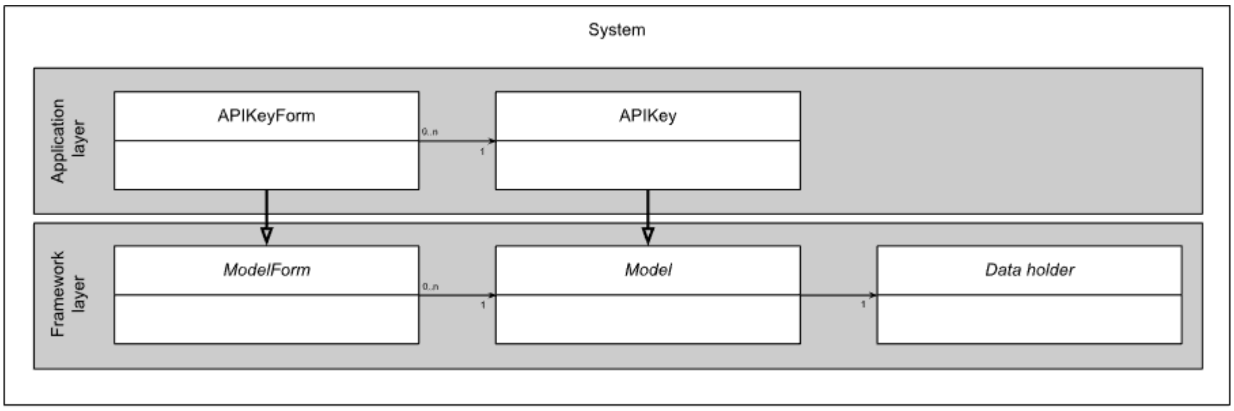


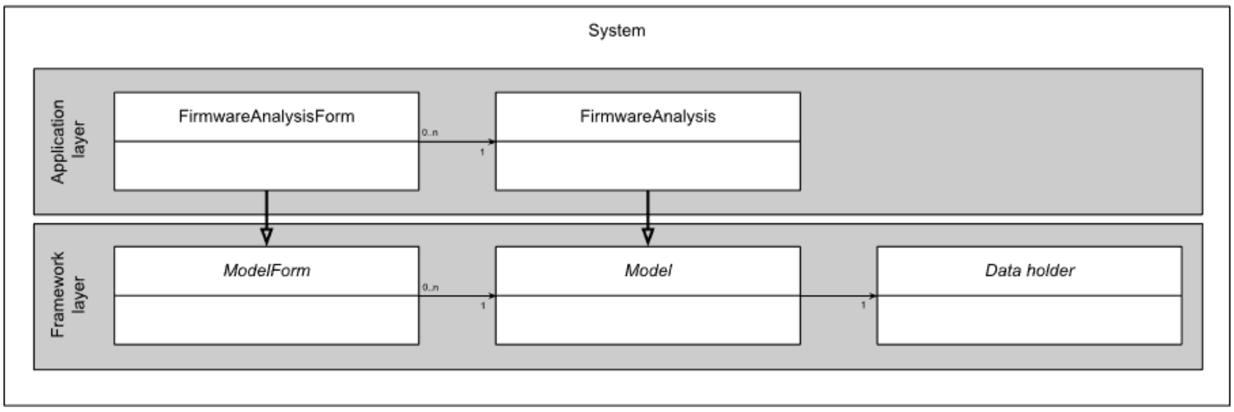


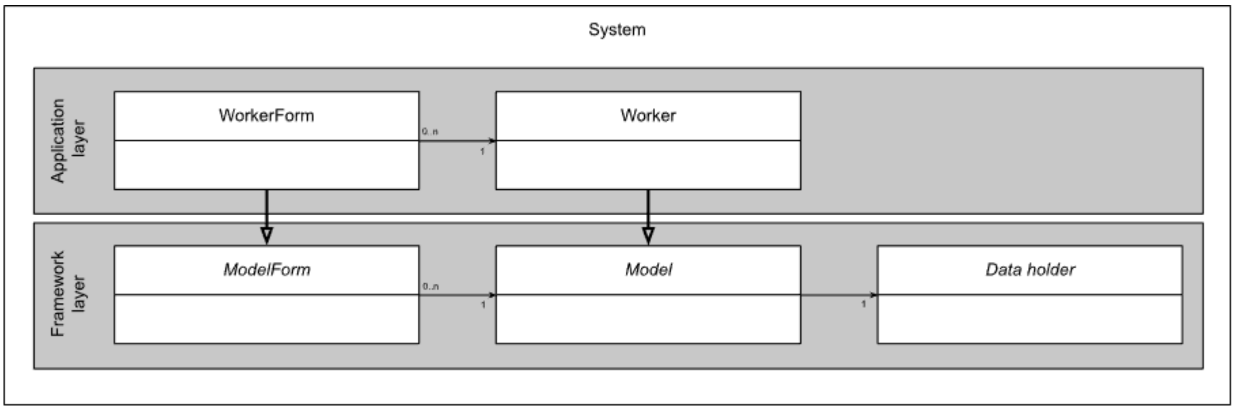




Code components







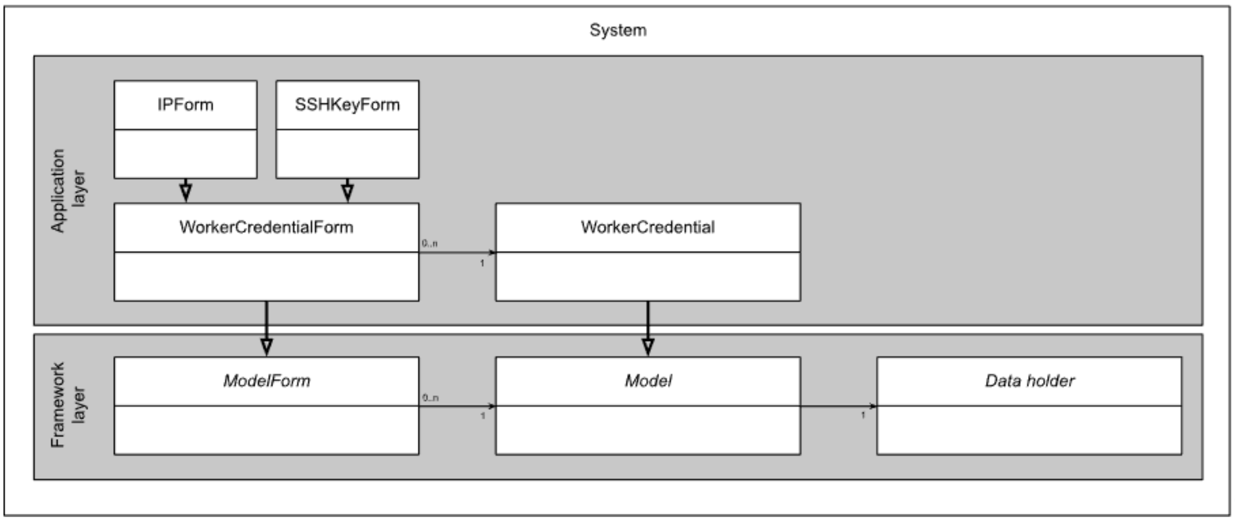


Diagram descriptions

Both diagrams are divided into four major topics:

1. API Keys: Generation and authorization
2. Firmware Analysis: Representation of arguments, status and result
3. Worker: Representation of worker nodes running firmware scans
4. Worker Credentials: Connection information such as IP, SSH key, …

Each of the major topics has a model representing its data and a corresponding form. The form serves both input (similar to a classic form) and output (a detailed view) purposes. For example, a user might want to create an *APIKey* using an *APIKeyForm*, and later retrieve its created API keys.

Similar to most web-applications, the architecture diagram can be divided into three tiers: The *presentation tier* includes everything the user directly interacts with, such as views, forms, etc. The *domain tier* includes all models representing our data. Finally, the *persistence tier* includes the persistent representation of the models. As Django uses Object-Relational Mapping (ORM), database queries are abstracted and thus not visible to the developer. For completeness, we represent those internal representations of models using the class *Data holder*.

As this project is implemented using Django, each code component diagram has two layers. The *application layer* represents all application specific classes and objects, such as the *APIKey* model. The *framework layer* represents all utility classes provided by Django, e.g. the *ModelForm*, representing input forms for models.