delegated to it—was to be discussed. To some extent, these discussions are still going on, as the technologies involved are evolving along with Akonadi, and the current approaches have yet to be validated in production use. At the time of this writing, there are agents feeding into Nepomuk and Strigi; these are separate processes that use the same API to the store as all other clients and resources. Incoming search queries are expressed in either XESAM or SPARQL, the respective query languages of Strigi and Nepomuk, which are also implemented by other search engines (such as Beagle, for example) and forwarded to them via DBUS. This forwarding happens inside the Akonadi server process. The results come in via DBUS in the form of lists of identifiers, which Akonadi can then use to present the results as actual items from the store to the user. The store does not do any searching or content indexing itself at the moment.

The API for the KDE-specific C++ access library took a while to mature, mostly because it was not clear from the start how type-independent the server would end up being and how much type information would be exposed in the library. By April 2007, it was clear that the way to extend the access library to support new types would be to provide so-called serializer plug-ins. These are runtime-loadable libraries capable of converting data in a certain format, identified by a mime type, into a binary representation for storage as a blob in the server and conversely, capable of restoring the in-memory representation from the serialized data. This is orthogonal to adding support for a new storage backend, for example, and the data formats used by it, which happens by implementing a new resource process (an agent). The responsibility of the resource lies in converting what the server sends down the wire into a typed, in-memory representation that it knows how to deal with, and then using a serializer plug-in to convert it into a binary datastream that can be pushed into the Akonadi store and converted back on retrieval by the access library. The plug-in can also split the data into multiple parts to allow partial access (to only the message body or only the attachments, for example). The central class of that library is called `Akonadi::Item` and represents a single item in the store. It has a unique ID, which allows it to be identified globally on the user's desktop and associated with other entities as part of semantic linking (for example, a remote identifier). This maps it to a source storage location, attributes, a data payload, and some other useful infrastructure, such as flags or a revision counter. Attributes and payload are strongly typed, and the methods to set and get them are templatized. `Akonadi::Item` instances themselves are values, and they are easily copiable and lightweight. `Item` is parameterized with the type of the payload and attributes without having to be a template class itself. The template magic to enable that is somewhat involved, but the resulting API is very simple to use. The payload is assumed to be a value type, to avoid unclear ownership semantics. In cases where the payload needs to be polymorphic and thus a pointer, or when there is already a pointer-based library to deal with a certain type of data (as is the case for `libkcal`, the library used for events and tasks management in KDE), shared pointers such as `boost::shared_ptr` can be used to provide value semantics. An attempt to set a raw pointer payload is detected with the help of template specialization and leads to runtime assertions.

The following example shows how easy it is to add support for a new type to Akonadi, provided there is already a library to deal with data in that format, as is frequently the case. It shows the