Models

Models are part of the MVC architecture. They are objects representing business data, rules and logic.

You can create model classes by extending yii\base\Model or its child classes. The base class yii\base\Model supports many useful features:

- Attributes: represent the business data and can be accessed like normal object properties or array elements;
- Attribute labels: specify the display labels for attributes;
- Massive assignment: supports populating multiple attributes in a single step;
- Validation rules: ensures input data based on the declared validation rules;
- Data Exporting: allows model data to be exported in terms of arrays with customizable formats.

The `Model` class is also the base class for more advanced models, such as Active Record. Please refer to the relevant documentation for more details about these advanced models.

> **Info:** You are not required to base your model classes on yii\base\Model. However, because there are many Yii components built to support yii\base\Model, it is usually the preferable base class for a model.

> **Warning:** Because by default all attributes of a model will be included in the exported array, you should examine your data to make sure they do not contain sensitive information. If there is such information, you should override `fields()` to filter them out. In the above example, we choose to filter out `auth_key`, `password_hash` and `password_reset_token`.

Best Practices

Models are the central places to represent business data, rules and logic. They often need to be reused in different places. In a well-designed application, models are usually much fatter thancontrollers.

In summary, models

- may contain attributes to represent business data;
- may contain validation rules to ensure the data validity and integrity;
- may contain methods implementing business logic;
- should NOT directly access request, session, or any other environmental data. These data should be injected by controllers into models;
- should avoid embedding HTML or other presentational code - this is better done in views;
- avoid having too many scenarios in a single model.

You may usually consider the last recommendation above when you are developing large complex systems. In these systems, models could be very fat because they are used in many places and may thus contain many sets of rules and business logic. This often ends up in a nightmare in maintaining the model code because a single touch of the code could affect several different places. To make the model code more maintainable, you may take the following strategy:

- Define a set of base model classes that are shared by different applications or modules. These model classes should contain minimal sets of rules and logic that are common among all their usages.
- In each application or module that uses a model, define a concrete model class by extending from the corresponding base model class. The concrete model classes should contain rules and logic that are specific for that application or module.

For example, in the Advanced Project Template, you may define a base model class `common\models\Post`. Then for the front end application, you define and use a concrete model class`frontend\models\Post` which extends from `common\models\Post`. And similarly for the back end application, you define `backend\models\Post`. With this strategy, you will be sure that the code in `frontend\models\Post` is only specific to the front end application, and if you make any change to it, you do not need to worry if the change may break the back end application.